EXAMINING APPLICATION COMPONENTS

TO REVEAL ANDROID MALWARE

THESIS

John B. Guptill, First Lieutenant, USAF

AFIT-ENG-13-M-19

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-13-M-19

EXAMINING APPLICATION COMPONENTS TO REVEAL ANDROID MALWARE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyberspace Operations

John B. Guptill, B.S.C.S.

First Lieutenant, USAF

March 2013

AFIT-ENG-13-M-19

EXAMINING APPLICATION COMPONENTS

TO REVEAL ANDROID MALWARE

John B. Guptill, B.S.C.S.
First Lieutenant, USAF

Approved:

Thomas E Dube   28 FEB 13

Maj Thomas E. Dube, PhD (Chairman)   Date

Rusty O. Baldwin   4 Mar 13

Rusty O. Baldwin, PhD (Member)   Date

Barry Mullins   28 Feb 13

Barry E. Mullins, PhD (Member)   Date

Grant A. Jacoby   12 MAR 13

COL Grant A. Jacoby, PhD (Member)   Date

AFIT-ENG-13-M-19

Abstract

Smartphones are becoming ubiquitous in everyday life and malware is exploiting these devices. Therefore, a means to identify the threats of malicious applications is necessary. This paper presents a method to classify and analyze Android malware through application component analysis.

The experiment parses select portions from Android packages to collect features using byte sequences and permissions of the application. Multiple machine learning algorithms classify the samples of malware based on these features. The experiment utilizes instance based learner, naïve Bayes, decision trees, sequential minimal optimization, boosted naïve Bayes, and boosted decision trees to identify the best components that reveal malware characteristics.

The best case classifies malicious applications with an accuracy of 99.24% and an area under curve of 0.9890 utilizing boosted decision trees. This method does not require scanning the entire application and provides high true positive rates. This thesis investigates the components to provide malware classification.

Table of Contents

List of Figures

List of Tables

Table Page

List of Acronyms

Acronym	Definition
AACS	Android Application Classification System
API	Application Programming Interface
AUC	Area Under Curve
BN	Bayesian networks
DT	Decision Trees
FN	false negative
FP	false positive
GPS	Global Positioning System
IB*k*	instance based learner
MMS	Multimedia Message Service
NB	Naïve bayes
OS	Operating System
PE	Portable Executable
ROC	Receiver Operating Characteristic
SAINT	Secure Application INTeraction
SDK	Software Development Kit
SIM	Subscriber Identity Module
SMS	Short Message Service
SMO	Sequential Minimal Optimization
SVM	Support Vector Machine
TN	true negative
TP	true positive
USB	Universal Serial Bus

EXAMINING APPLICATION COMPONENTS TO REVEAL ANDROID MALWARE

I. Introduction

THE classification of malicious applications is a difficult problem. The creators of malware do not openly disclose the methods that they use to exploit systems. They are constantly finding new methods to obfuscate their malicious programs. The possibility of malware infecting mobile devices and smartphones affect multiple facets of users. Users do not use smartphones only for making phone calls, but also for shopping, and banking. For this reason, protecting the devices requires a means to detect malware on the Android platform.

The Android Operating System (OS) is a platform on both smartphones and tablets. Applications are available for Android to perform many different tasks related to gaming, social networking, news, and productivity. A popular location to download applications from is the Google Play service formerly known as the Google Market [22]. If consumers use Google Play as the exclusive distributor of applications, they should not need to worry about downloading malicious applications that may cause harm to devices or personal information. Yet Google Play has distributed malicious applications in the past [4]. In addition to Google Play, other sources for applications include downloading directly from third party markets or websites. When many options are available as sources of applications, a method to detect malicious applications is necessary [51].

The approach in this experiment is to utilize n-grams to classify Android applications. An n-gram is a sequence of bytes from a data source of length n. The experiment extracts the n-grams using three sources found inside of the Android application package. These three files describe the contents and intentions of the application. This research uses these three files rather than examining the entire application package. In addition to the *n*-gram extraction, this research also considers application permissions. This research only selects a portion of the features using calculations from the information gain of the samples. Lastly, six different machine learning algorithms classify the samples using the selected features.

1.1 Research Goals and Hypothesis

The overall goal of this research is to classify an Android application as either malicious or benign. The term *benign* describes the applications identified as "not malicious" according to current anti-virus tools. Goals associated with this research are to determine the best performing machine learning algorithm and feature source to accomplish the overall goal. Aspects of these schemes include the following:

1. Determine which file or feature source provides the best classification. The experiment extracts three files from each application that are information rich concerning the applications use and its attributes. The experiment also utilizes the permissions of the application as an additional feature source. The best approach has the highest successful detection rates.

2. Determine which classifier gives the highest accuracy. The experiments use four different base classifiers to evaluate each of the files. Boosting applies to two of the classifiers to increase their performance. Overall, the experiment has six classifiers, four non-boosted and two boosted variants.

3. Determine if specific *n*-grams classify applications more accurately than other *n*-grams. An *n*-gram or set of *n*-grams may give rise to specific phrases that may be useful in classifying the intent of an application.

A feature source containing the ranked *n*-grams from all three files used should provide successful classification. From the individual feature sources, the hypothesis is that the `classes.dex` may have the best results for classification because it contains the source code for the application. This file can be large in some cases and may not provide the fastest approach. The fastest approach may come from the permission badging. Badging is command to extract the application permissions and does not require any additional unpacking.

1.2 Research Contributions

This research contributes to the field of Android malware classification. This research parses files in the application into *n*-grams to find an effective and fast means for classification. This research also compares the effectiveness of permissions and *n*-gram feature sources in Android malware classification. This research also uses chaining of *n*-grams found in the feature set to find clues that would lead to the intention of the malicious applications. This research makes the following contributions to the field of Android malware classification and *n*-gram studies:

- Identify a feature source in the applications that classifies at a high rate of accuracy in Section 4.4.8.
- Test the effectiveness of multiple classifiers on detecting Android malware in Section 4.4.
- Analyze sequences of *n*-grams to identify malware intention in Section 4.3.1

1.3 Summary

This section introduces the problem of classifying malicious Android applications. It describes the goals associated with the experiment and describes the initial approach to the problem. Lastly, the section describes the research contributions.

II. Literature Review

2.1 Overview

This chapter discusses the portions of Android architecture and applications pertinent to this research. This section also reviews the previous work of others in the field of mobile malware classification and security. An understanding of the internals of an Android application is necessary to understand where possible exploits may occur and to select the best feature sources. This literature review focuses on research in the last five years related to the Android environment and the use of machine learning algorithms. Research includes both dynamic and static analysis of applications. Lastly, this chapter reviews classifiers and machine learning principles used in the experiment. These classifiers are Naïve Bayes, Instance Based Learner, Decision Trees, and Sequential Minimal Optimization (SMO).

2.2 Android Architecture

Android is a mobile Operating System (OS) developed to run on devices such as mobile phones and tablets. The applications use a modified Java Virtual Machine called the Dalvik Virtual Machine. The Dalvik Virtual Machine operates on devices with lower resources than personal computer such as mobile phones. Application developers use the Java programming language during development. The compiler converts the source code to Dalvik byte code to run on the device. The Android OS runs on a Linux-based architecture. The applications are in `.apk` packages downloaded from Google Play or other locations. These packages contain files that run in a Dalvik Virtual Machine. Tools exist that convert the Dalvik code into Java classes [46]. The `.apk` application packages also contain an `AndroidManifest.xml` file, which contains information for the Android OS. This information includes permissions to interact with the OS and capabilities that the application requires. In the `.apk` file is the `classes.dex` file, which contains the compiled application code. This portion contains the Dalvik byte code that runs on the Android device [21]. Zip utilities such as `gzip` or `unzip` can unpack the compressed Android applications.

To use aspects of the device to access the Internet or send Short Message Service (SMS) messages, the Android application must declare permissions in the `AndroidManifest.xml file`. The permissions tell the system that the application has access to pertinent system Application Programming Interface (API) calls. The default permissions use the following syntax, `android.permission.RESOURCE`. Custom permissions for API's follow the `namespace.permission.RESOURCE` syntax. Not all APIs require permissions to run. The Android Software Development Kit (SDK) [20] provides a utility, `aapt`, to dump information about applications. This utility is able to dump the permissions for an application as seen in Figure 2.1.

```
uses-permission:  android.permission.ACCESS_WIFI_STATE
uses-permission:  android.permission.WRITE_SMS
uses-permission:  android.permission.RECEIVE_BOOT_COMPLETED
uses-permission:  android.permission.VIBRATE
uses-permission:  android.permission.READ_SMS
uses-permission:  android.permission.RECEIVE_SMS
uses-permission:  android.permission.SEND_SMS
uses-permission:  android.permission.DISABLE_KEYGUARD
uses-permission:  android.permission.READ_CONTACTS
uses-permission:  android.permission.WRITE_CONTACTS
uses-permission:  android.permission.INTERNET
uses-permission:  android.permission.ACCESS_NETWORK_STATE
uses-permission:  android.permission.READ_PHONE_STATE
uses-permission:  android.permission.CALL_PHONE
```

Figure 2.1: Example Android permissions from `aapt` output

At build time, the packing process adds the `.dex` files and the manifest files to the `.apk` package. In addition to these, the packing process adds all other resources to the package. The `resources.arsc` contains the resource table of the resources contained in the application [21].

2.3 Android Malware

This paper will explore detection of Android malware applications. Sources of malware include self-propagating malware on mobile devices that spread via Voice over Internet Protocol or Multimedia Message Service (MMS) [17]. Other sources of infection include via Internet, synchronizing to a computer, or peer-to-peer to other cell phones [24]. Additionally, there are proofs of concept to drain cellphone batteries through SMS [40].

Felt, et al. analyzed 46 samples of malware for three different platforms, which included 18 Android samples [16]. Among these samples, the most common malicious applications collected user information and sending premium rate SMS messages. Felt specified seven different incentives for mobile malware:

- novelty and amusement,
- selling user information,
- stealing user credentials,
- making premium-rate calls and SMS,
- SMS spam,
- search engine optimization, and
- ransom.

F-Secure discovered the first virus on mobile phones, which was on a Palm device in 2000 [30]. One of the first mobile viruses was *Cabir*, which infected multiple platforms and spread via Bluetooth. The virus would prompt nearby users asking if they wanted to receive a message and infect the device when the user accepted the message. Other early versions of viruses masqueraded as games. The *Metal Gear* virus appears to be a video game released to other platforms that is available to mobile devices. Once the user installs the game, the malware disables the tools on the phone that would be able to remove the program and sends another virus via Bluetooth to other users.

Mobile viruses that exploit the SMS and Bluetooth services are possible [8]. In 2004, Dagon, Martin, and Starner predicted multiple malicious uses for applications on mobile devices [11]. Such methods may include information theft, such as stealing data from the device or find the location of the user by broadcasting their location. Unsolicited information may also be a threat by placing advertisements on the device that the user did not intend to appear. Denial of Service attacks may also appear that will not allow the user to use a specific service on the device.

Zhou and Jiang have more than 1,200 samples of Android malware found in the Google Play market and other third party markets [50]. The malware in their analysis have the following payloads:

- privilege escalation - application gains higher privileges on the phone than necessary to perform otherwise unauthorized actions,
- remote control - application allows a central server to control the device without informing the user,
- financial charge - application uses services that charge the users without their knowledge, and
- information collection - application gathers information on the user such as phone numbers and other account information.

2.4 Related Work

As more malicious applications appear on the Android platform, multiple researchers have attempted to identify these applications [6, 9, 14, 42, 44, 45, 49, 53]. Methods include using static and dynamic analysis of the application and measuring performance on the device. Security is important because personal information stored on the device and applications may request more permissions than necessary.

2.4.1 Detecting Android Malware Through Dynamic Analysis.

In 2012 Shabtai, et al. created a framework to identify malware based on behavior [45]. The framework analyzed system metrics, such as the processor consumption, network usage, number of processes and battery usage. In this study, they selected sets of 10, 20, and 50 features out of a possible 88 features. They attempted to use classifiers such as *k*-means, logistic regression, histograms, decision trees, Bayesian networks (BN), and Naïve Bayes. At the time of this experiment, Shabtai et al. were unable to locate true malicious applications. They had to create their own for this experiment. Their dataset only included four malicious applications. Through these experiments they determined that Naïve Bayes and Logistic Regression were the best performing classifiers.

Another attempt at classifying applications on Android is "Crowdroid" [9]. This approach utilizes dynamic means to classify applications. This novel approach uses the technique of crowd sourcing to complete the task. This method utilizes an application available on the Google Android market that monitors Linux Kernel system calls and sends the logs from multiple users to a central server to cluster using *k*-means to determine if the application is malicious. Each test utilizes benign versions and malicious versions of the same applications and classifies the trace of system calls. Due to the lack of known malicious Android applications, they wrote three malicious programs and had two known malicious applications from other sources. Through recording multiple call traces of applications, the approach was able to classify 100% traces of their own malware, but only identified the actual malware traces 85% of the time using clustering.

The Kirin security service performs certification on applications to mitigate the threat of malware [14]. The service looks at the permissions and services that the application requests and compares them to a set of defined rules. These rules include requesting to record audio or location tracking via Global Positioning System (GPS). Upon comparing these permissions to the rules, the application would tell the user if the application failed a safety check. They tested 311 applications from the Android Google Market and 12 of these applications gave warnings of being malicious. The applications themselves may not be malicious, but still allow the application more permissions than necessary for the application's use.

2.4.2 Detecting Android Malware Through Static Analysis.

Static methods provide a method to analyze the application without requiring installation, but may not provide as much insight as dynamic analysis. Shabtai, et al. attempted to classify Android applications through static analysis [44]. The experiment's goal is to classify between tools, games and not malicious applications. In this experiment, they looked at the different elements in Android Manifest XML files, the `classes.dex` file and aspects of the `.apk` file, including file size and methods. To classify the applications, they utilized classifiers such as decision trees, naïve Bayes, Bayesian networks, Partial Decision Trees (PART), boosted Naïve bayes (NB), boosted Decision Trees (DT), random forest, and voting Feature Intervals. The experiment showed that the Boosted DT classifiers and Bayesian Networks were the best classifiers tested.

Schmidt, et al. used static analysis for malware detection on Android [42]. Using a dataset of 240 malicious samples, they extracted the names of functions and calls inside already installed executables. They classified the application using three different classifiers: PART, Prism, and nearest neighbor algorithms. The system utilizes collaboration with other devices to improve the classification.

Wu, et al. built the framework DroidMat to detect malware on Android [49]. DroidMat looks at elements in the `AndroidManifest.xml` file and API calls to classify applications. DroidMat used 238 malicious applications from Contagio Mobile [33] and 1,500 benign applications from the Android Market. Wu used a k-means algorithm to classify the samples. The framework was able to classify samples at a 97.87% accuracy rate using this method.

Zhu, et al. reviewed the text of an application description and observed the permissions that the application requests [53]. The dataset in this experiment included 5,685 applications downloaded from the Android Market. For the experiment, they selected 23 specific permissions and then selected dominant words in the application description. Zhu then classifies the words as positive or negative based on the action the permission allows. Positive words are ones that would be normal permission behavior. Negative words describe the permission use in a manner in an unsafe manner. They used this method to identify if the application is abnormal based on the presence of positive and negative words. The framework was able to get a 90% true positive rate, but had a 30% false positive rate. Meaning that they were able to properly identify 90% of the malware, but improperly classify 30% of the benign samples as malicious.

2.4.3 Combination of both Static and Dynamic Analysis.

Bläsing, et al. proposed a sandbox environment to both statically and dynamically classify applications on the Android platform [6]. In the static analysis, Bläsing is searching decompiled applications for the usage of `System.getRuntime().exec`, permissions, and interprocess communications. The dynamic analysis aspect of the research records system calls. Through their experiments Bläsing, et al. used 150 applications from the Google Market. They use a self-written `fork` bomb application to test malware samples. The sandbox environment successfully detects the `fork` bomb application. The sandbox environment does no other malicious tests due to the lack of known malicious applications at the time.

2.4.4 Malware Detection on Non-Android Mobile Platforms.

SmartSiren was another approach to detect malicious applications on smartphones [10]. The framework was for a Windows Mobile device rather than an Android platform. The SmartSiren approach reports activities to a central server to perform the detection. Agents on the devices send reports anonymously to a proxy through SMS traces. The platform explores the various infection methods such as Bluetooth, MMS, Internet, Universal Serial Bus (USB), and peripheral connections. SmartSiren calculates the effectiveness of detection through statistical monitoring of the behavior.

Bose, Hu, Shin and Park proposed a behavioral malware detection framework on the Symbian devices [7]. They utilize behavior signatures that are events generated by the application and specifically looking at single or a collection of possible malicious events. One behavior that the framework monitors to generate signatures are the Symbian API calls. The proposed system examines samples that only affect the Symbian OS. The framework classified malware utilizing Support Vector Machine (SVM)'s with an accuracy of 96%.

Another attempt to detect malicious applications on the Symbian platform was from Liu, Yan, Zhang, and Chen [31]. The research utilized battery consumption of the mobile device for detection. They assumed that malicious applications would cause more battery usage than normal applications. In this implementation, they used Decision Trees, Neural Networks, and Linear Regression to classify the presence of malicious activity by monitoring battery consumption. The implementation successfully identified the presence of *Cabir* and *FlexSpy* which are viruses for the Symbian platform.

An additional attempt to detect Symbian Malware was Schmidt, Clausen, Camtepe, and Albayrak by looking at application function calls [43]. In their experiment, they used 33 malicious applications and 49 benign applications. From these applications, only 2,620 unique function calls existed. Using Centroid Machines, Naïve Bayes, and Binary SVM to classify the programs using the function calls as the features. Through their research, they noted that most malicious applications on Symbian used Bluetooth to spread. The results of their experiments showed that the Centroid Machines had the highest accuracy, but the Binary SVM had the highest true positive detection rate.

Table 2.1 shows a summary of the experiments that perform machine learning in order to classify mobile applications. Each of the experiments are looking at different features and classifiers than the experiment in this paper. Experiments of others that share features or classifiers with this experiment are in bold. For example, Shabtai looks at the `classes.dex` and the `AndroidManifest.xml`, but does not perform analysis via *n*-grams.

2.4.5 Platforms for Privacy Security.

Enck, Octeau, McDaniel, and Chaudhuri studied more than 1,000 Android applications [13]. They developed a decompiler for Android applications to analyze the source code to find different security vulnerabilities. Through this study, they found that many applications had potential security holes.

TaintDroid developed by Enck, et al. tracks passed messages between Android applications to detect possible malicious applications [12]. TaintDroid tracks possible privacy leaks and not specifically malware classification. TaintDroid tracks information flow between applications by adding markers to different variables. These markers are able to tell if an application is passing privacy-sensitive information to another application with little overhead. They were able to identify 30 applications from the Google Market that pass private information. The applications passed information such as the phone number and Subscriber Identity Module (SIM) card serial number to other applications.

Park, et al. proposed a framework to detect unauthorized access to root privileges [37]. The main purpose is to combat specific types of malware that attempt privilege escalation on Android such as Droid-KungFu, DroidDream, and GingerMaster. The framework

Table 2.1: Comparison of experiment features and classifiers

Shabtai 2010 [44]	Features	APK Features, `classes.dex`, `AndroidManifest.xml`
	Classifiers	**Decision Trees** , **Naïve Bayes**, Bayesian Networks, PART, Boosted BN, **Boosted DT**, Random Forest, Voting Feature Interval
Andromaly [45]	Features	Application Level, Operating System, Scheduling, Memory, Keyboard, Network, Hardware, Power
	Classifiers	*k*-means, Logistic Regression, Histograms, **Decision Trees**, Bayesian Networks, **Naïve Bayes**
Crowdroid [9]	Features	System metrics from multiple users
	Classifiers	*k*-means
SmartSiren [10]	Features	Bluetooth, MMS, Internet, USB, Peripheral Connections
	Classifiers	Statistical Monitoring
VirusMeter [31]	Features	Battery Consumption
	Classifiers	Logistic Regression, Neural Networks, **Decision Trees**
Schmidt 2009 [42]	Features	Function Calls
	Classifiers	PART, Prism, Nearest Neighbor
Schmidt 2009[43]	Features	Function Calls
	Classifiers	Centroid Machine, **Naïve Bayes**, **SVM**
DroidMat [49]	Features	**Permissions**, Function Calls
	Classifiers	K-means, Nearest Neighbor, **Naïve Bayes**
Zhu 2012 [53]	Features	**Permissions**, Intention Words
	Classifiers	**Naïve Bayes**
Bose 2008 [7]	Features	System Events, API calls
	Classifiers	**SVM**
Guptill	Features	*n*-grams, Permissions
	Classifiers	Instance Based, Naïve Bayes, Decision Trees, SVM(SMO), Boosted DT, Boosted NB

prevents privilege escalation through a *pWhitelist* and a *Criticallist*. The *pWhitelist* is a list of applications that have access to `root` privileges while the *Criticallist* is a list of resources that even `root` users cannot modify. This prevents applications that gain root from using critical resources. This framework was capable of preventing an application from gaining `root` privileges.

2.4.6 Protection Through Managing Permissions.

Felt, et al. created Stowaway which maps API calls to permissions that an application uses [15]. Stowaway determines which applications have unnecessary permissions. They used Stowaway on 940 applications from the Android Market. Their tests showed that one-third of the applications tested had permissions that were not necessary..

Nauman, Khan, and Zhang proposed a framework which would allow for enforcement of permissions on Android [34]. The motivation for this problem is that the user must allow all permissions when installing an application. They developed Poly, which extends the current Android application installer. Poly gives the user the ability to block or constrain to a number of uses for specific permissions at install time. They do not describe the impact of Poly on the application when an application attempts to exercise the denied permission.

Ongtang, McLaughlin, Enck, and McDaniel presented Secure Application INTeraction (SAINT) which manages application permissions both at install time and during runtime [36]. The SAINT installer gathers the permissions and the package's signatures and checks these permissions with a preset policy. When the applications run, SAINT continues to interact with the applications. The system monitors when applications start, when applications receive messages from other applications, when applications access content providers, and when the applications needs to access system APIs.

Zhou, Zhang, Jiang, and Freeh proposed a framework to add additional permission specifications to protect sensitive information on the Android platform [52]. An example of sensitive information is GPS location or user information. The framework provides a manager for application permissions on the device. The user is able to change privacy settings for each application, giving the user the capability to provide true data, mock data, or no data for that specific setting.

Beresford, Rice, Skehin, and Sohan at the University of Cambridge proposed a modified version of the Android OS to add another layer of security [5]. The main change to the OS is to the Package Manager service and performs checks on API calls and permissions. The OS, Mock-Droid, provides false data to applications to prevent them from getting access to sensitive information. The framework allows the user to restrict access so the application is able to run, but not with sensitive information. Permissions that can be restricted include GPS data, Internet connectivity, or access to other phone functions.

Hornyack, et al. built a system that adds privacy protection such that applications are unable to send out protected information [26]. Protected information may include phone state, contacts, or SMS or MMS messages. The additional security blocks the applications from getting to the sensitive information. This removes the primary motivations of Android malware as discussed in §2.3. After running their tests on applications, they saw blocking these messages caused side effects such as removing advertisements, causing the application to be less functional, or completely breaking the application in 34% of the applications.

2.4.7 Security through different markets.

The main route of Android application distribution is through the market Google Play. Google Play has hosted malicious applications in the past. A study by Zhou et al. shows that at least 32 malicious applications were on the site in May to June 2011 [51]. Google may have removed these applications, but the possibility to infect devices is still present.

DroidRanger is a system developed at North Carolina State University to detect malicious applications on markets [51]. The system performs both permission-based filtering and behavioral matching to detect known malware. In the permission based filtering, DroidRanger filters out applications that do not use permissions present in malicious applications. The second step looks at the rules within the `AndroidManifest.xml`, the API calls invoked by the application, and the application structure. Another part of DroidRanger attempts to detect unknown Android malware through heuristics. These heuristics include flagging if the application is running native code or dynamically loading remote binary code. Overall, the system detected 211 malicious applications from five different markets with a total of 204,040 samples. This method only used two heuristics to identify malicious applications and more heuristics exist to detect different types of malware.

Multiple solutions exist to improve security from the distribution point rather than from the device. These solutions would attempt to find malicious applications and remove them from the market. Announced in February 2012, Google began to develop a Bouncer which is an automated service that will approve applications before distributing them to the public [32]. Google also released an application verification service with the release of Jellybean Android 4.2 that allows application validation on the device to ensure they are not malicious. The service uploads information such as the SHA1 value, version, and the URL associated with the application to the cloud for evaluation. Jiang at North Carolina State University tested this new service utilizing the same set of malicious applications used in this experiment [27]. Googles service only detected 193 of the 1260 known malicious applications. The Google service may only check against known malware samples in their database and may not identify previously unknown samples that DroidRanger could.

2.5 *n*-grams in Malware Detection

Kolter and Maloof utilized machine learning to detect malicious Windows Portable Executable (PE) files [29]. They extracted n-grams from the Windows PE files to classify the applications. To select the best *n*-grams as features they calculated information gain to the counts of unique n-grams in each classifying set. The following information gain equation calculates the value:

$$IG(j) = \sum_{v_j \in \{0,1\}} \sum_{C \in \{C_i\}} P(v_j, C) log \frac{P(v_j, C)}{P(v_j)P(C)} \tag{2.1}$$

where the class C either malicious or harmless, and v_j represents the count of *n*-grams in the class set. They utilize the following classifiers: the instance-based learner, naïve Bayes, SVM, decision trees, and boosted variants except for naïve Bayes. They use Weka, a machine learning suite from The University of Waikato [25] to train and test the classifiers. They collected 1,971 benign executables and 1,651 known malicious executables. From their results they calculate the areas under the ROC curves shown in Table 2.2.

Table 2.2: Kolter Maloof experiment results

Method	AUC
Boosted Decision Trees	0.9958±0.0024
SVM	0.9925±0.0033
Boosted SVM	0.9903±0.0038
IBk, $k = 5$	0.9899±0.0038
Boosted Naïve Bayes	0.9887±0.0042
Decision Trees	0.9712±0.0067
Naïve Bayes	0.9366±0.0099

They found that this method was highly effective in detecting malicious applications. In the paper they calculate the Receiver Operating Characteristic (ROC) curves using the posterior probability of the negative class which were the benign samples. Overall, they found that the boosted decision trees were the best classifier tested using this method with an area under the curve of 0.9958±0.0024.

Abou-Assaleh, et al. also use n-grams to detect malicious Windows PE files [1]. The data set in this experiment involves 25 malicious and 40 benign samples. In this experiment, they vary the size of the n-grams and the length of the profile. In training, they detect high accuracy when the size of the n-gram is greater or equal than three. The final experiment performs 3-fold cross validation and detects an average accuracy of 98%.

The experiment explores the use of n-grams in malicious code detection, but only uses a small set of samples.

2.6 Classifiers

When discussing the topic of classifying items in relation to machine learning, a background in the topic of each classifier is necessary. Other experiments discussed in the literature review utilized different classifiers. The research in this thesis only uses select number of classifiers.

2.6.1 Naïve Bayes.

Naïve Bayes is a classifier that assumes that each of the features, F_i, are conditionally independent [28]. Let $p(C)$ be the probability of the class and $p(F_i|C)$ be the probability of a feature for the given class C. The classifier selects the class with the highest probability in the following equation:

$$p(C)\prod_i p(F_i|C). \tag{2.2}$$

This classifier works best on datasets where the features are conditionally independent. Datasets with dependent features cause the effectiveness of Naïve Bayes to drop [48].

2.6.2 IBk.

The instance based learner is a lazy algorithm [2]. The instance based learner saves all of the training samples and compares the test samples to each of the members of the training set until it finds the closest match. Weka implements the instance based algorithm as a k-nearest neighbor classifier. The Weka implementation sets Euclidean distance as the default distance algorithm [48].

2.6.3 Sequential Minimal Optimization.

An implementation of SVM in the Weka suite is SMO [38]. The SVM maps the samples on multiple dimensions. The SVM and SMO algorithms classify the samples by calculating a separator between the two classes and then maximizing the width of this margin [41]. To solve the problem of multiple dimensions, SMO calculates the maximization by splitting the problem into smaller parts. Each problem consists of optimizing two multipliers in order to maximize or minimize the solution. The algorithm solves the smallest first and adds these to the overall optimization. The classifier uses either a Gaussian or a polynomial kernel to map the data.

2.6.4 Decision Trees.

A decision tree is a classifier that generates a tree [39]. The decision tree assigns a prediction to a sample when it traverses the tree and reaches a leaf node. The algorithm traverses the tree starting at the root of the tree. The tree consists of decision nodes, which is a test on an attribute to split the population of samples. The node evaluates each sample to check the next branch to traverse based on the value of the node. The classifier assigns the sample a label based on the value of the end leaf node.

2.6.5 Boosting Algorithms.

Boosting is a method to improve the performance of classifiers, such as decision trees or naïve Bayes [18]. In `AdaBoost.M1`, a classifier runs multiple times to reduce the error. The first iteration all of the instances have the same weight. As the iterations continue, the boosting process adds weights to the instances from the results of the classifier runs. The weight of the instances may change depending on the output of the classifier [48]. Once all the runs are complete, the boosting algorithm returns the result.

2.7 Summary

In summary, this chapter reviews the background in the field of Android malware research. The understanding of the Android application structure shows where exploits may occur. In order to carve out the research that this research performs, Table 2.3 shows others research and the fields that apply to this experiment. The experiment in this paper covers the use of *n*-grams, permissions, and static analysis in order to classify Android applications. Though this paper discusses the topic of dynamic malware analysis for Android, the experiment only uses static analysis. Lastly, this section discussed the different classifiers in this experiment.

Table 2.3: Review of Android malware classification

Research Field	Source	Guptill Research
n-grams	[1, 29]	•
Permissions Studies	[15, 34, 36, 52]	•
Dynamic Analysis	[9, 13, 14, 37, 43, 45, 51, 53]	
Static Analysis	[42, 44, 49, 51, 53]	•

III. Experimental Design

3.1 Overview

This chapter discusses the experimental design to accomplish the classification of the Android applications. The problem definition describes the approach and goals for this experiment. This chapter also presents the various aspects of the system under test such as the system boundaries, the system services, workload, and system parameters. Finally, the section describes the evaluation technique for the experiment.

3.2 Approach

The approach in this research is to use machine learning algorithms with different feature sources to detect malicious Android applications. These feature sources include the following:

- `classes.dex`,
- `resources.arsc`,
- `AndroidManifest.xml`, and
- application permissions.

The approach of this research is the use of *n*-grams on the first three feature sources. This research utilizes *n*-grams because gathering them is efficient and represents sequences of instructions or revealing patterns in a file. Three files always exist at the root of the Android application `.apk`. These files are the `classes.dex`, `resources.arsc`, and the `AndroidManifest.xml`. The `resources.arsc` file contains a list of resources for the application. The `classes.dex` contains the classes that represent the Dalvik bytecode of the application. The `AndroidManifest.xml` contains the intent, activities, and permissions for the application. This research extracts these three files from each

Android application and scans for four byte *n*-grams with a one byte sliding window as shown in Figure 3.1. The sliding window allows every four-byte sequence to have representation. This experiment selects sequential *n*-grams as features they represent an entire phrase and not just the initial four bytes. This experiment uses four byte length *n*-grams which is the same as Kolter and Maloof [29].

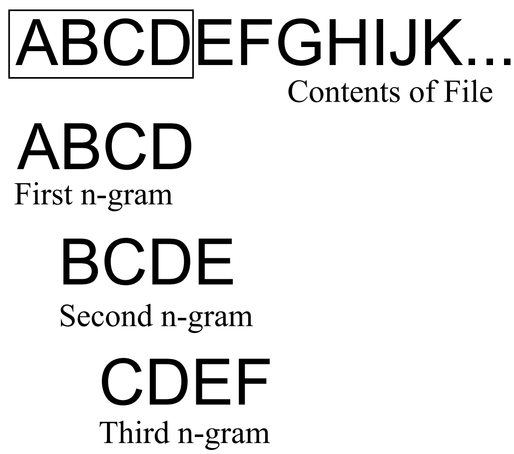

Figure 3.1: *n*-gram extraction

This research utilizes information gain to select the top features from each source. The information gain calculation assigns a value that indicates if a feature is more representative of a class. The information gain value is calculated with the following equation:

$$IG(j) = \sum_{v_j \in \{0,1\}} \sum_{C \in \{C_i\}} P(v_j, C) log \frac{P(v_j, C)}{P(v_j)P(C)}, \tag{3.1}$$

where v_j is the value of the feature j and C represents the class i. The $P(v_j, C)$ is the probability of a feature given a specific class [29].

To calculate the information gain the feature extractor counts the presence of each n-gram only once per sample. For instance, if a feature source contains multiple instances of a specific *n*-gram, the *n*-gram is only counted once. The experiment consists of two sample sets, a set of known malicious applications and a set of benign applications. The *n*-gram count for each sample set is a sum of all *n*-gram counts for each application. Information gain uses the *n*-gram counts from the class sets to calculate an information gain value for each *n*-gram. This research combines the three file sources into an additional feature source. For each application the *n*-gram extractor also counts an aggregate of the unique *n*-grams for all three feature sources to create a “combined” feature source.

From the four *n*-gram based feature sources, the extractor sorts the *n*-grams by their information gain values. Kolter and Maloof tested different numbers of features with classifiers and found that 500 features provided the best results for their selected classifiers [29]. This experiment only uses the top 500 *n*-grams from each feature source for the selected features since this research uses the same classifiers as Kolter and Maloof.

In addition to looking at the effectiveness of n-grams, the framework analyzes the permissions with the same classifiers. The Android SDK provides a tool, `aapt` that returns the permissions of each application. `aapt` does not require the overhead of unpacking the application. Other researchers also use permissions as features in their work [49, 53]. The framework leverages this tool to extract the permissions and create two new sets of features. One set of features uses the extended permission name. An example of this is `android.permission.INTERNET` for the Internet access permission. The other set uses the tail of the permission. This case only uses `INTERNET` to identify the internet access permission. The framework also gives each permission an information gain value and ranking as with the n-grams.

The framework runs the features, `AndroidManifest.xml`, texttresources.arsc, `classes.dex`, combined *n*-grams, full permissions, and permission tail through seven classifiers using Weka [25]. The experiment uses the following classifiers:

- instance based learner,
- naïve Bayes,

- decision trees,
- SMO,
- boosted naïve Bayes, and
- boosted decision trees.

3.3 System Boundaries

The system under test is the Android Application Classification System (AACS). The AACS implements machine learning algorithms to detect malicious applications. A set of 1,260 malicious and a set of 16,577 benign Android samples serve as the workload. The system parameters are the selected classifiers to classify the applications and the feature sources. The components of the system under test are the Sample Parsing, the Information Gain Calculation and Sorting, Feature Extraction and the Sample Classification. The component under test is the Sample Classification. The Sample Parsing component parses the samples for *n*-grams from the file feature sources and the permissions and counts the unique features in each sample set. The Information Gain Calculation and Sorting calculates the information gain values of the *n*-grams in each feature source. This component then ranks the *n*-grams and permission based on the resulting information gain values. The Feature Extraction component selects and extracts the top 500 *n*-grams from each file based feature sources. There are only 1,083 unique full permissions and 557 permission tails in the sample sets. The Feature Extraction component uses all permissions to extract features. The Sample Classification component uses the selected classifiers to classify the samples using the feature sources. The system under test outputs the accuracy, the Area Under Curve (AUC) of the ROC curves, and the classification results.

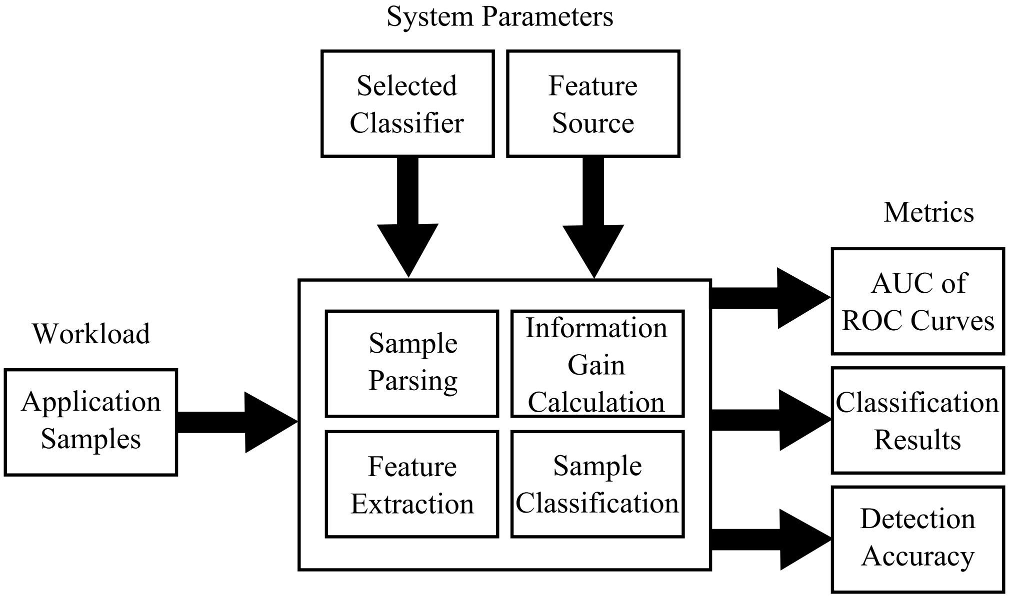

Figure 3.2: System under test

3.4 System Services

The primary service of the system under test is the classification of Android applications. The underlying services that the system provides: extracting *n*-grams from the applications, calculating information gain for each *n*-gram, selecting the highest n-grams based on information gain, extracting selected *n*-grams from the applications, and the eventual classification of the applications. the applications for the n-grams, ranking based on the information gain, collecting features based on the rank and the eventual classification of the applications. The output of the overall system service is if an application is malicious or not malicious.

3.5 Workload

The workload is a set of Android applications collected from different sources. The set consists of 1,260 Android malware samples and 16,577 assumed non-malicious samples. The experiments use the samples on all classifiers and parsing methods. The size of Android applications varies from a few kilobytes to multiple megabytes.

Developers build the applications for different versions of the Android OS depending on the available Android SDK. Each new version of the SDK may add new security features or tools. The first two charts in Figure 3.3 show the version of the OS the developers used to create the samples in the experiment. The third chart is the market distribution of Android OS version in January 2013 [23]. The malware samples are from August 2010 to September 2011, while the benign samples are from July 2012 to August 2012. Even though the samples are from different years they are still built with the same Android versions. The chart only represents 937 of the 1,260 malicious samples and 16,198 of the 16,577 benign samples. The build date information is not available for all of the applications in the sample sets.

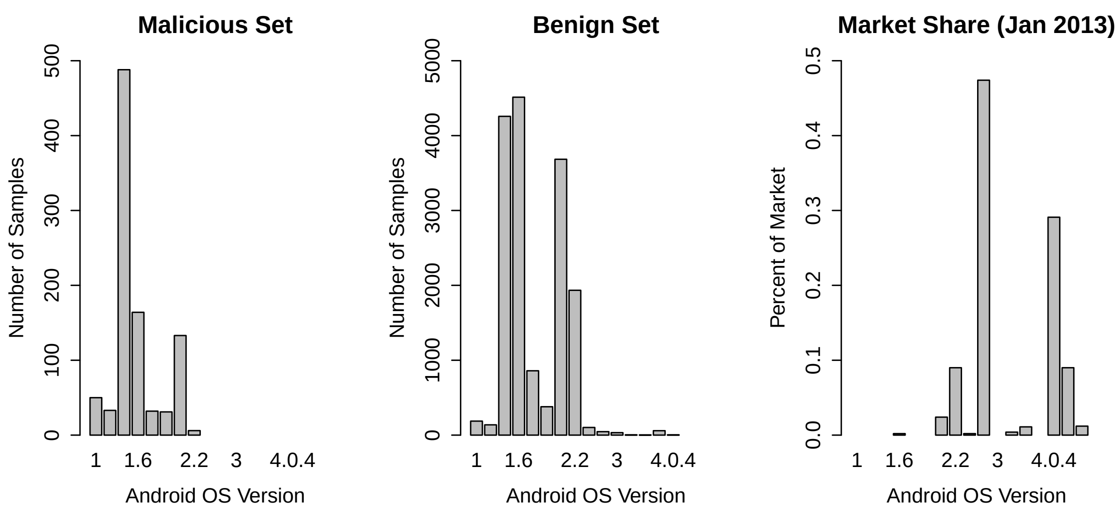

Figure 3.3: Sample set Android version and market share

The set of malicious applications are from the Android Malware Genome Project at North Carolina State University [50]. This set contains 1,260 different malicious applications from 49 different malware families. The families and the count of the application in each family are in Table 3.1. VirusTotal antivirus service independently validated malicious applications from NCSU as malware [47]. VirusTotal uses multiple antivirus utilities to check if a sample is malicious.

Table 3.1: Malicious samples

Android Malware			
Family Name	Number of Samples	Family Name	Number of Samples
ADRD	22	GingerMaster	4
AnserverBot	187	GoldDream	47
Asroot	8	Gone60	9
BaseBridge	122	GPSSMSSpy	6
BeanBot	8	HippoSMS	4
Bgserve	9	Jifake	1
CoinPirate	1	jSMSHider	16
CruseWin	2	KMin	52
DogWars	1	LoveTrap	1
DroidCoupon	1	NickyBot	1
DroidDeluxe	1	NickySpy	2
DroidDream	16	Pjapps	58
DroidDreamLight	46	Plankton	11
DroidKungFu1	34	RoughLemon	2
DroidKungFu2	30	RoughSPPush	9
DroidKungFu3	309	SMSReplicator	1
DroidKungFu4	96	SndApps	10
DroidKungFuSapp	3	Spitmo	1
DroidKungFuUpdate	1	Tapsnake	2
Endofday	1	Walkinwat	1
FakeNetflix	1	YZHC	22
FakePlayer	6	zHas	11
GamblerSMS	1	Zitmo	1
Geinimi	69	Zsone	12
GGTracker	1		

The benign samples come from various third-party Android markets across the Internet. Table 3.2 indicates the sources and number of samples for the benign set of applications. The sources for the benign samples may contain duplicates between each set. The combined count is the total number of unique samples from both sources. There are 16,577 total benign samples in this experiment. This set may potentially contain malicious samples. VirusTotal validated the samples as benign in October 2012. VirusTotal provides reports for each sample uploaded from multiple anti-virus products. If at least one product from VirusTotal identifies a sample as possibly malicious, the experiment does not include the sample in the benign set.

Table 3.2: Sources for benign samples and counts

Source	Number of Samples Collected
nduoa.com [35]	14,105
APKTOP [3]	2,964
Combined	16,577

3.6 Performance Metrics

As the framework processes samples it classifies them as either malicious or non-malicious. The framework knows the true classification of each sample. The system evaluates the number of true positives, false positives, true negatives, and false negatives from of each classifier and parsing method. The experiment utilizes bootstrapping to calculate 95% confidence intervals for each of the performance metrics.

3.6.1 Classification Results.

One primary metric output from the system is the classification result. This output includes the average true positive (TP), false negative (FN), true negative (TN), and false positive (FP) rates for each classifier. The equations for the rates are in Table 3.3. In addition to this, the classification result report the average numbers of classified samples in each of the previously mentioned categories.

Table 3.3: Classification rates for results

$\frac{\text{TP}}{\text{TP} + \text{FN}}$	$\frac{\text{FP}}{\text{FP} + \text{TN}}$
$\frac{\text{FN}}{\text{TP} + \text{FN}}$	$\frac{\text{TN}}{\text{FP} + \text{TN}}$

3.6.2 ROC curves.

The ROC curve is an additional metric to measure the effectiveness of the feature source with the classifier. The ROC is a plot of true positive rate against the false positive rate of the classifier. The AUC of the ROC gives a value to the represent the performance of the classifier. In this experiment, the positive result is the malicious classification.

3.7 System Parameters

The system parameters are the selected classifier and the feature source for the applications. The following machine learning algorithms are the classifiers:

- instance based learner (IBk),
- naïve Bayes,
- decision trees,
- SMO,
- boosted naïve Bayes, and
- boosted decision trees.

These classifiers are the same machine learning algorithms that Kolter and Maloof [29] use in their experiments. Their experiments utilize *n*-grams in a similar fashion to the parsing methods in this experiment. Kolter and Maloof extract *n*-grams from Windows PE files. This experiment extracts *n*-grams from specific locations in the Android application. This experiment does not use the boosted SMO classifier that Kolter and Maloof use in their experiment due to the time to complete the experiments.

The other set of system parameters are the feature sources. The feature sources are the n-grams extracted from the `classes.dex`, `resources.arsc`, and the `AndroidManifest.xml`. An additional *n*-gram based feature source is the aggregate *n*-grams from these three file sources. The last two feature sources are the permissions from the applications represented as the full permission and the permission tail.

This experiment does not consider the specific parameters to the hardware running the evaluations. The intention of this research is to classify applications correctly before they reach the device. This research is hardware independent.

3.8 Evaluation Technique

The experiment is the measurement of a real system. The system classifies actual Android applications. The experiments run on a virtual machine running Ubuntu 12.04 LTS with 128 GB of memory allocated. The virtual machine resides on a Dell PowerEdge R810 with 512 GB of memory running Ubuntu 12.04 LTS. The system runs inside of a virtual machine due to the presence of malicious samples. The algorithms to parse and perform feature collection are in C++ version 4.6. The classifiers are part of the Weka suite [25].

3.9 Experimental Design

A full factorial design evaluates the effectiveness of each factor of classifier and feature source combination. The six selected classifiers and six parsing methods result in 36 different experiments. Each run utilizes 10 k-fold cross-validation. The experiment is run with 10 replications to show that consistent results with respect to false positives and true positives given by each run as well as ROC curve values.

3.10 Summary

This research presents a system that classifies Android applications as malicious or non-malicious using machine learning algorithms. The system uses actual Android applications collected from various sources. The system uses n-grams from three separate sources and permissions to attempt to classify the malicious applications. These include looking at the `classes.dex`, `resources.arsc`, and the `AndroidManifest.xml` file in the `.apk` package. The classifiers in this experiment are the instance based learner, naïve Bayes, decision trees, and SMO. This experiment uses a boosting algorithm on naïve Bayes and the decision trees. The experiments run as measurements on a real system while the algorithms to perform the parsing with C++ and classification with Weka. The goal is to determine which feature source and classifier provides the highest detection rate with the lowest number of false positives.

IV. Results and Analysis

4.1 Overview

This chapter presents the results of the Android malware classification problem as described in Chapter 3. The experiment consists of three parts to classify the applications. The first aspect of the research is to extract the features from the application through n-gram parsing and dumping permissions of the files. The second part involves calculating the information gain of each feature and selecting the top 500 features. The last section of the experiment is classifying the samples using the features and classifiers selected.

4.2 Data Extraction

The first aspect of the experiment is determining the features necessary to allow for an effective classification of the Androfid applications. The experiment extracts two types of features from the sample applications. The framework utilized three files in the application to extract n-grams. Additionally, the framework dumps the permissions of the samples to generate the second set of features. The permission set consists of two types of permissions, the full permission and the permission tail.

4.2.1 N-gram Extraction and Analysis.

The initial types of features extracted in this experiment are *n*-grams. The framework parses three files extracted from the application for *n*-grams. The files are the `AndroidManifest.xml`, `classes.dex`, and the `resources.arsc`. The method to parse the *n*-grams is a sliding window format of four-byte chunks from the three designated files. The parsing algorithm also tracks an aggregate of the unique *n*-grams of the three files from the Android applications. Of the Android sample set, each Android application has an average of 233,016 unique *n*-grams in the `classes.dex` file that is much larger than the average 1,696 or 18,156 *n*-grams in the `AndroidManifest.xml` and `resources.arsc` sources. The `classes.dex` has more unique *n*-grams because the file size is much larger than both the `AndroidManifest.xml` and the `resources.arsc`. Additionally, the `classes.dex` consists of byte code while the `AndroidManifest.xml` contains specific formats for permissions, intents and other information about the application. The benign set contains a total of 238,674,477 unique *n*-grams and the malicious set of application contains 28,779,383 unique *n*-grams.

4.2.2 Permission Extraction.

In a similar fashion to the *n*-grams, the parser counts the permissions for each sample. Just as with the *n*-grams, the permissions have separate counts for each sample set. This process does not require the extractor to unpack the application. A utility provided with the Android SDK [20], `aapt`, is able to dump the permissions in plain text for each sample. The parser counts the full permission name and the tail of the permission. Both datasets contain 1,082 unique full permissions and 556 unique permission tails phrases.

4.3 Information Gain results

The next component is the information gain calculation. This calculates the information gain after the parser extracts the *n*-grams from the sample sets. Each feature source has a count of unique *n*-grams in each sample set. The information gain formula uses counts from the malicious and benign set to calculate an information gain value for each *n*-gram. The last part of this process ranks the *n*-grams based on their information gain value. This experiment only uses the top 500 *n*-grams as features for the classifiers.

The top *n*-grams from the individual feature sources do not share any unique *n*-grams with the other feature sources. The feature sources may share *n*-grams, but there is no overlap with the top *n*-grams. The information gain values of the *n*-grams in those files may not be high enough to be in the top 500 *n*-grams of their feature source. The *n*-grams found in the top 500 *n*-grams of the combined files contains 479 of the 500 top *n*-grams from the `classes.dex` source.

Additionally, 19 of the *n*-grams in the combined source are from the top 500 *n*-grams of the `AndroidManifest.xml` file. These 19 *n*-grams are in the top 22 *n*-gram values for the `AndroidManifest.xml` file. Since the combined feature source includes *n*-grams from all three *n*-gram sources, the presence of mutual *n*-grams changes the value in the combined source. The addition of the `AndroidManifest.xml` n-grams to the combined file may increase the effectiveness of the combined feature source.

In Figure 4.1 the information gain values from the `AndroidManifest.xml` start higher than the other feature sources. The top information gain values in `AndroidManifest.xml` are higher than the other feature sources. By the 11th *n*-gram, the values of the `AndroidManifest.xml` drop below the information gain values of the `classes.dex` and the combined feature sources. The `resources.arsc` almost has a linear change in information gain for the top n-grams.

4.3.1 Chaining of n-grams.

Observation of the top *n*-grams shows an overlap of the bytes between *n*-grams. For, example as seen in Figure 4.2 the top two *n*-grams in hexadecimal representation from `classes.dex` have three bytes that overlap. The first three bytes of the first *n*-gram (`0x9D83EFBC`) are the same as the last three bytes of the second *n*-gram (`0xE69D83EF`). The fact that the *n*-grams are from the same source, have the same information gain and parsed using a sliding window of one byte, they may be part of the same byte sequence. The resulting sequence is `0xE69D83EFBC`. The other top ten *n*-grams from each source exhibit similar patterns as seen in Figure 4.2.

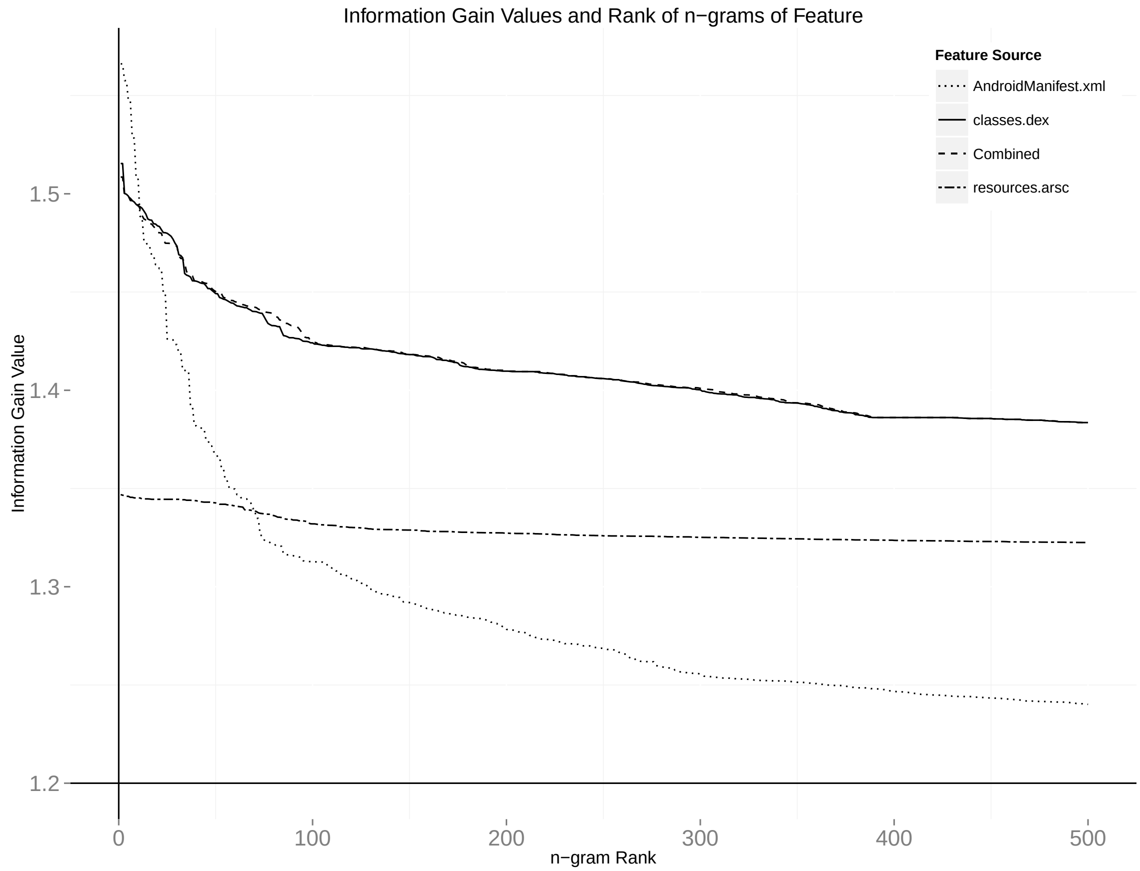

Figure 4.1: Information gain value results

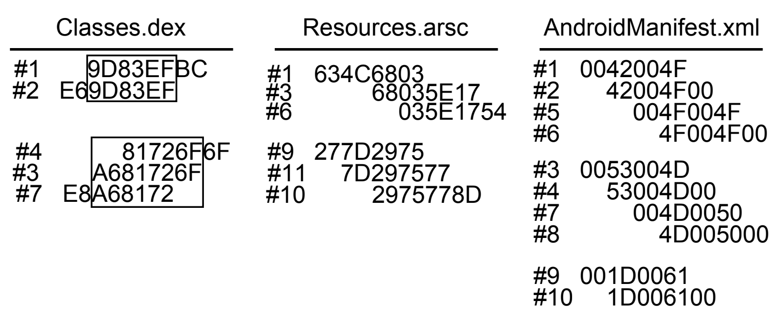

Figure 4.2: Overlapping top 10 *n*-grams from each source

The *n*-gram sequences chained together may provide insight into the reasons for the selected *n*-grams in the top 500 for having high information gain values. To find all likely sequences a script finds all *n*-grams within each source with three overlapping bytes. A single *n*-gram may have multiple *n*-grams that can create a sequence. For example, the *n*-gram `0x74E69D83` is able to connect to `0xE69D8300` or `0xE69D83E9`. The result is two chained sequences. When the chaining continues, the possibility exists of one of the `n`-grams repeating if it matches the three-byte condition of chaining. In this case, the problem of chaining becomes a cyclic graph. The chaining algorithm ignores repeats and attempts a different matching *n*-gram.

Every other byte in the *n*-grams from the `AndroidManifest.xml` is zero, such as `0x004F0042` or `0x4F00F200`. The high number of zeros in the `AndroidManifest.xml` *n*-grams increases the number of overlapping *n*-grams. To control the number of possible chains, the chaining algorithm checks number of times the *n*-gram occurs in the malicious set. If the count is within a limit then the algorithm appends the *n*-gram to the end of the sequence. The algorithm attempts to make the longest chains possible. The main goal of the limit is to minimize the algorithm from creating too many `AndroidManifest.xml` chains. When the count limit is set to ten, the number of chains produced from the `classes.dex` is the same as when it is not set. The number of sequences in the `resources.arsc` is 559 when set and 563 when not set.

The chaining algorithm creates 509 unique chains from top 500 n-grams for the `classes.dex` source. Table 4.1 shows the first 23 chains produced from the `classes.dex` source. The *n*-gram position in the table is the rank of the information gain value of the first *n*-gram in the chain. The ASCII representations of the *n*-gram chains are in the third column. The table only shows the bytes that have ASCII readable characters. If the hexadecimal for the byte is beyond the readable ASCII characters then column omits the character. The ASCII representation of the chain shows both the phrase `roo` and `oot` are highly ranked chains. Since the algorithm only checks for an overlap of three bytes and not two bytes, the algorithm does not generate the chain `root`. The `.dex` format has a strings section in ASCII, but the encoding of the rest of the file is in LEB128 (Little-Endian Base 128) [19]. The fourth column shows the average number of times that the *n*-grams that create the chain occur in the malware sample set. The *n*-grams that make up the `root` phrase occur in the malware sample set 453 and 487 times. According to Zhou [50], who provided the samples, 36% or 453 of the samples attempt to use `root` level exploits. This number is close to the average times the phrase occurs in the malware.

Table 4.1: ASCII interpretation of top *n*-gram chains for `classes.dex`

n-gram Position	n-gram chain	ASCII of chain	Average counts in Malware Set
1	0x9D83EFBC	[□□□□]	527
2	0xE69D83EFBC	[□□□□□]	527
3	0xA681726F6F	[□□roo]	467
4	0x81726F6F	[□roo]	467
5	0x3D5A8405	[=Z□□]	464
6	0x9D83E7AEA1	[□□□□□]	465
7	0xE8A681726F6F	[□□□roo]	467
8	0x9990E6898D	[□□□□□]	464.5
9	0x1E5A5A1C05	[□ZZ□□]	464
10	0x6F74E69D83E7AEA1	[ot□□□□□□]	466.2
11	0x74E69D83E7AEA1	[t□□□□□□]	466
12	0x83E7AEA1	[□□□□]	465
13	0x6F6F74E69D83E7AEA1	[oot□□□□□□]	466.333
14	0x8CE68E88	[□□□□]	464
15	0x1FE99C80	[□□□□]	448
16	0x001FE99C80	[□□□□]	448
17	0x1C2F7368	[□□/sh]	449
18	0x0B73737469	[□ssti]	449.5
19	0x152F6461	[□/da]	521
20	0xA18CE68E88	[□□□□□]	464

The chaining algorithm produces 4,839 chains from the top 500 *n*-grams from the `AndroidManifest.xml`. The n-grams contain alternating null bytes (`0x00`). The `AndroidManifest.xml` file is in a binary XML format with strings in Unicode. The top 24 chains from the `AndroidManifest.xml` file show phrases that are common in malware that Zhou identifies [50]. Zhou has identifies that 83.3% or 1,125 of the malware from the sample use the BOOT_COMPLETED system event. The *n*-gram chains from positions 1, 2, 7, 8, 11,12, 21 and 22 in Table 4.2 all have about the same number of occurrences in the malware set. The chaining algorithm does not create the full phrase of BOOT_COMPLETED. The sections of the phrase, T_, ET, and TE, occur in more manifest permissions than the other portions of the phrase BOOT_COMPLETED. The frequency of these sections in other permissions may change the overall information gain value of the *n*-gram and not include it in the top 500 *n*-grams. Malicious applications utilize the BOOT_COMPLETED event to start as soon as the device has completed booting. The chain from position 3, SMS, is also common among malicious applications. The algorithm generates chains that are permissions found in malware samples.

Table 4.2: ASCII interpretation of top *n*-gram chains for AndroidManifest.xml

n-gram Position	n-gram chain	ASCII of chain	Average counts in Malware Set
1	0x0042004F004F00	[B O O]	1,125.5
2	0x42004F004F00	[B O O]	1,127
3	0x0053004D005300	[S M S]	894
4	0x53004D0053004D	[S M S M]	894
5	0x004F004F00	[O O]	1,130
6	0x4F004F004F	[O O O]	1,130
7	0x004D0050004C004500	[M P L E]	1,083.67
8	0x4D0050004C004500	[M P L E]	1,086.6
9	0x001D006100	[a]	923
10	0x1D006100	[a]	923
11	0x0050004C004500	[P L E]	1,091
12	0x50004C004500	[P L E]	1,092.67
13	0x0059005F004200	[Y _ B]	906
13	0x0059005F0041004E00	[Y _ A N]	919.333
14	0x59005F004200	[Y _ B]	903.667
14	0x59005F0041004E00	[Y _ A N]	920.6
15	0x004500440000002C00	[E D ,]	1,158.67
16	0x4500440000002C00	[E D ,]	1,158.6
17	0x0052005900	[R Y]	651
18	0x52005900	[R Y]	651
19	0x0043005400	[C T]	988
20	0x43005400	[C T]	988
21	0x004F004D0050004C004500	[O M P L E]	1,083.75
22	0x4F004D0050004C004500	[O M P L E]	1,083.71

4.3.2 Top Permissions with Information Gain.

There are only 556 unique permission strings with permission tails and 1,082 unique permissions with the full permission string. Table 4.3 shows the top ten permissions with the highest information gain values. The top four permissions relate to SMS. The previous section shows that the phrase SMShas a high information gain value in the `AndroidManifest.xml` file. According to Zhou, 45.3% of the malicious applications send and receive SMS messages [50]. Additionally, the permission, RECEIVE_BOOT_COMPLETED, is in the `AndroidManifest.xml` file as seen in the previous section.

Table 4.3: Top 10 permissions by information gain value

Permission	Information Gain Value
`android.permission.READ_SMS`	1.584209753
`android.permission.WRITE_SMS`	1.535876854
`android.permission.SEND_SMS`	1.373848696
`android.permission.RECEIVE_SMS`	1.371761714
`android.permission.RECEIVE_BOOT_COMPLETED`	1.369654341
`android.permission.WRITE_APN_SETTINGS`	1.343876281
`android.permission.READ_PHONE_STATE`	1.305217899
`android.permission.WRITE_CONTACTS`	1.297509651
`android.permission.ACCESS_WIFI_STATE`	1.28330626
`android.permission.CALL_PHONE`	1.267492607

4.4 Classification Results

In the previous sections, this experiment creates six different feature sets. Utilizing the suite Weka from the University of Waikato, six different machine-learning algorithms classify each of the feature sets [25]. The classifiers in this experiment are the same classifiers as Kolter and Maloof with the exception of the boosted SMO [29]. The next section will discuss the results of the classification experiments.

4.4.1 `resources.arsc` Results.

The `resources.arsc` feature source performs the worst among the other feature sources in the experiment. Table 4.4 shows the comparison of the `resources.arsc` to the other feature sources with each classifier. The table indicates that in each test there is a statistical improvement of using the other feature sources over `resources.arsc`. The mean AUC for `resources.arsc` across classifiers is 0.6412 which is significantly less than the other feature sources.

Table 4.4: ROC area under the curve comparison to `resources.arsc`

Dataset	`resources.arsc`	Combined	`classes.dex`
IB*k*	0.6738(0.6692–0.6782) -	0.9808(0.9791–0.9824) ∘	0.9608(0.9587–0.9629) ∘
Naïve Bayes	0.6140(0.6087–0.6187) -	0.9114(0.9085–0.9140) ∘	0.8605(0.8564–0.8642) ∘
Boosted NB	0.6189(0.6142–0.6238) -	0.9458(0.9432–0.9481) ∘	0.8351(0.8311–0.8393) ∘
Decision Trees	0.6568(0.6520–0.6613) -	0.9556(0.9530–0.9580) ∘	0.9458(0.9430–0.9488) ∘
Boosted DT	0.6817(0.6773–0.6858) -	0.9751(0.9731–0.9769) ∘	0.9550(0.9523–0.9575) ∘
SMO	0.6021(0.5986–0.6058) -	0.8159(0.8119–0.8201) ∘	0.8139(0.8100–0.8182) ∘
Dataset	**`AndroidManifest.xml`**	**Full Permission**	**Permission Tail**
IB*k*	0.9841(0.9825–0.9857) ∘	0.9806(0.9790–0.9822) ∘	0.9807(0.9790–0.9822) ∘
Naïve Bayes	0.9306(0.9279–0.9331) ∘	0.9222(0.9192–0.9247) ∘	0.9240(0.9213–0.9266) ∘
Boosted NB	0.9751(0.9733–0.9766) ∘	0.9497(0.9471–0.9523) ∘	0.9483(0.9457–0.9507) ∘
Trees	0.9634(0.9605–0.9660) ∘	0.9568(0.9537–0.9598) ∘	0.9565(0.9532–0.9596) ∘
Boosted DT	0.9890(0.9877–0.9902) ∘	0.9789(0.9770–0.9806) ∘	0.9792(0.9774–0.9808) ∘
SMO	0.9419(0.9390–0.9447) ∘	0.8301(0.8246–0.8350) ∘	0.8232(0.8183–0.8278) ∘

- comparison dataset
∘ statistically significant improvement
• statistically significant degradation

The SMO classifier identifies all of the benign samples in `resources.arsc` for every run and fold in the experiment. This identification leads to a false positive rate of 0.0 and a true negative rate of 1.0 as seen in Table 4.5. The decision boundary for the SMO successfully classifies all of the benign samples, but this also includes 79.57% of the malicious samples. Boosted decision tree and decision tree classifiers identify 347.0 of the malicious samples. The SMO is able to identify on average 257.3 samples from 10 runs.

Table 4.5: Confusion matrix rates for `resources.arsc`

Confusion Matrix Rates with 95% CI

	TPR	FPR	TNR	FNR
Naïve Bayes	0.2071(0.2006–0.2140)	0.005477(0.005132–0.005867)	0.9945(0.9941–0.9948)	0.7928(0.7855–0.7996)
Boosted Decision Trees	0.2753(0.2676–0.2836)	0.003625(0.003341–0.003933)	0.9963(0.9960–0.9966)	0.7246(0.7165–0.7327)
Boosted Naïve Bayes	0.2071(0.2005–0.2139)	0.005477(0.005106–0.005878)	0.9945(0.9941–0.9948)	0.7928(0.7852–0.8001)
SMO	0.2042(0.1974–0.2113)	0.0(0.0–0.0)	1.000(0.0–0.0)	0.7957(0.7886–0.8032)
Instance Based	0.2745(0.2669–0.2829)	0.003112(0.002852–0.003371)	0.9968(0.9966–0.9971)	0.7254(0.7168–0.7339)
Decision Trees	0.2753(0.2669–0.2834)	0.003625(0.003347–0.003922)	0.9963(0.9960–0.9966)	0.7246(0.7165–0.7329)

All of the classifiers in this experiment return low true positive rates. The best true positive rate is 0.2754 with a confidence interval of 0.2672 to 0.2833 with the boosted decision trees. The low classification rate may be due to only a few malware using the `resources.arsc` in abnormal ways. Figure 4.3 shows distribution of the frequency of the number of positive features in samples. The number of positive features is the number of *n*-gram features in each sample. A sample may have from 0 to 500 positive features. A majority of the samples on the benign side to Figure 4.3 have almost no positive features. While the malicious side of Figure 4.3 is a bimodal distribution. The samples have either almost no positive features or most of the features. The number of samples in the peaks of these histograms matches the classification results of the classifiers.

Since the number of samples with high positive features is about the same as the number of samples the classifiers identify as malicious, a relationship may exist. The malicious dataset contains 222 samples with at least 450 positive features. All of the samples are from only two families of malware of the 49 families in the dataset. Of the 222 samples, 139 are from the `AnserverBot` family and 83 are from the `BaseBridge` family. Key elements of the `resources.arsc` files of the malware from these families

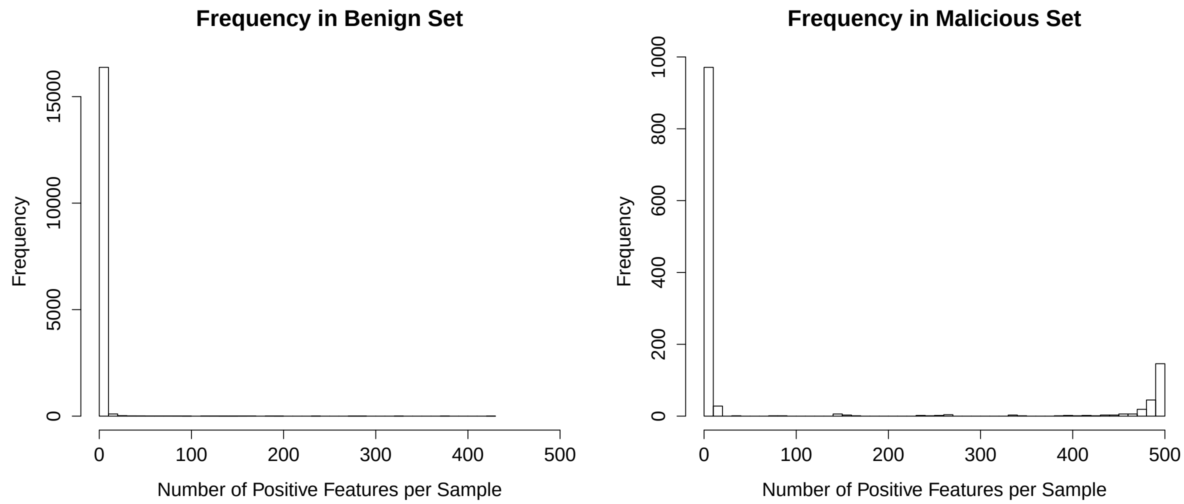

Figure 4.3: Frequency of positive features of datasets for `resources.arsc`

must be different such that the *n*-grams in these files are not in other `resources.arsc` files. The features that this experiment selects perform well for these two families, but fail to detect other malicious applications.

4.4.2 `classes.dex` Results.

Observing the `classes.dex` feature source in comparison in Table 4.6 to the other feature sources `classes.dex` has a statistically significant improvement over `resources.arsc`. All of the other feature sources perform better than the `classes.dex` with the exception of the combined and permission based feature sources. These sources perform better, but not with statistical significance.

Observing the distribution of the samples and the number of positive features, `classes.dex` has a similar distribution as the `resources.arsc` as seen in the Figure 4.4. In the case with `classes.dex` the benign sample with the most positive features has 61 positive features. Looking at the samples with the highest number of positive features the

Table 4.6: ROC area under the curve comparison to `classes.dex`

Dataset	`classes.dex`	Combined	`AndroidManifest.xml`
IB*k*	0.9608(0.9587–0.9629) -	0.9808(0.9791–0.9824) ∘	0.9841(0.9825–0.9857) ∘
Naïve Bayes	0.8605(0.8564–0.8642) -	0.9114(0.9085–0.9140) ∘	0.9306(0.9279–0.9331) ∘
Boosted NB'	0.8351(0.8311–0.8393) -	0.9458(0.9432–0.9481) ∘	0.9751(0.9733–0.9766) ∘
Decision Trees	0.9458(0.9430–0.9488) -	0.9556(0.9530–0.9580)	0.9634(0.9605–0.9660) ∘
Boosted DT	0.9550(0.9523–0.9575) -	0.9751(0.9731–0.9769) ∘	0.9890(0.9877–0.9902) ∘
SMO	0.8139(0.8100–0.8182) -	0.8159(0.8119–0.8201)	0.9419(0.9390–0.9447) ∘
Dataset	**`resources.arsc`**	**Full Permission**	**Permission Tail**
IB*k*	0.6738(0.6692–0.6782) •	0.9806(0.9790–0.9822) ∘	0.9807(0.9790–0.9822) ∘
Naïve Bayes	0.6140(0.6087–0.6187) •	0.9222(0.9192–0.9247) ∘	0.9240(0.9213–0.9266) ∘
Boosted NB'	0.6189(0.6142–0.6238) •	0.9497(0.9471–0.9523) ∘	0.9483(0.9457–0.9507) ∘
Decision Trees	0.6568(0.6520–0.6613) •	0.9568(0.9537–0.9598)	0.9565(0.9532–0.9596)
Boosted DT	0.6817(0.6773–0.6858) •	0.9789(0.9770–0.9806) ∘	0.9792(0.9774–0.9808) ∘
SMO	0.6021(0.5986–0.6058) •	0.8301(0.8246–0.8350)	0.8232(0.8183–0.8278)

- comparison dataset
∘ statistically significant improvement
• statistically significant degradation

top 309 all belong to the `DroidKungFu3` family of malware. In the malicious set there are only 309 samples of `DroidKungFu3`.

Investigating further into the number of positive features and the samples associated with the counts there is a pattern of features and their families. In the malicious sample set, 469 samples have more than 61 positive features. Of the 469 malicious samples from this set, they are all members of the `DroidKungFu` families of malware. There are 473 samples in the `DroidKungFu` families. This shows that a majority of the features from `classes.dex` are found in the `DroidKungFu` families.

Even though the samples with high feature counts are only with the `DroidKungFu` family of malware, the feature source still has high true positive rates as a classifier in comparison to the `resources.arsc` as seen in Table 4.7. The true positive rates for both the naïve Bayes and the boosted naïve Bayes are 0.3734 which is smaller than the other

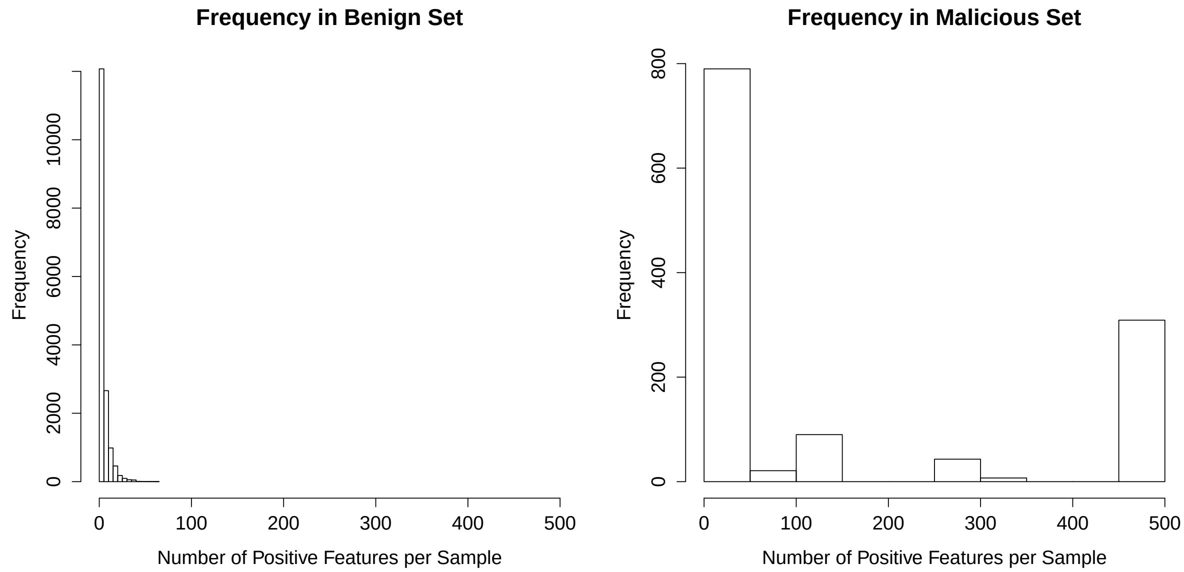

Figure 4.4: Frequency of positive features of datasets for `classes.dex`

rates in the `classes.dex` feature source. The best true positive rate is from the boosted decision trees with a rate of 0.81831.

Table 4.7: Confusion matrix rates for `classes.dex`

Confusion Matrix Rates with 95% CI

	TPR	FPR	TNR	FNR
Naïve Bayes	0.3734(0.3657–0.3818)	0.0004222(0.0003316–0.0005160)	0.9995(0.9994–0.9996)	0.6265(0.6183–0.6350)
Boosted Decision Trees	0.8131(0.8067–0.8194)	0.003963(0.003623–0.004295)	0.9960(0.9956–0.9963)	0.1868(0.1803–0.1930)
Boosted Naïve Bayes	0.3734(0.3648–0.3813)	0.0004222(0.0003301–0.0005222)	0.9995(0.9994–0.9996)	0.6265(0.6179–0.6352)
SMO	0.6279(0.6198–0.6353)	0.0001206(6.968e-05–0.0001667)	0.9998(0.9998–0.9999)	0.3720(0.3645–0.3792)
Instance Based	0.7754(0.7684–0.7825)	0.004578(0.004250–0.004914)	0.9954(0.9951–0.9957)	0.2245(0.2173–0.2317)
Decision Trees	0.7952(0.7873–0.8027)	0.006490(0.006104–0.006890)	0.9935(0.9931–0.9939)	0.2047(0.1972–0.2116)

4.4.3 `AndroidManifest.xml` results.

The `AndroidManifest.xml` has a higher ROC AUC for each classifier in comparison to the other feature sources as seen in Table 4.8. The instance based learner performs better, but not with significance between the combined and the permission based feature sources. The ROC AUC is higher with the instance based learner with 0.9841 in comparison to the boosted decision trees with 0.9634.

Table 4.8: ROC area under the curve comparison to `AndroidManifest.xml`

Dataset	`AndroidManifest.xml`	Combined	`classes.dex`
IB*k*	0.9841(0.9825–0.9857) -	0.9808(0.9791–0.9824)	0.9608(0.9587–0.9629) •
Naïve Bayes	0.9306(0.9279–0.9331) -	0.9114(0.9085–0.9140) •	0.8605(0.8564–0.8642) •
Boosted NB'	0.9751(0.9733–0.9766) -	0.9458(0.9432–0.9481) •	0.8351(0.8311–0.8393) •
Decision Trees	0.9634(0.9605–0.9660) -	0.9556(0.9530–0.9580)	0.9458(0.9430–0.9488) •
Boosted DT	0.9890(0.9877–0.9902) -	0.9751(0.9731–0.9769) •	0.9550(0.9523–0.9575) •
SMO	0.9419(0.9390–0.9447) -	0.8159(0.8119–0.8201) •	0.8139(0.8100–0.8182) •
Dataset	`resources.arsc`	Full Permission	Permission Tail
IB*k*	0.6738(0.6692–0.6782) •	0.9806(0.9790–0.9822)	0.9807(0.9790–0.9822)
Naïve Bayes	0.6140(0.6087–0.6187) •	0.9222(0.9192–0.9247) •	0.9240(0.9213–0.9266)
Boosted NB'	0.6189(0.6142–0.6238) •	0.9497(0.9471–0.9523) •	0.9483(0.9457–0.9507) •
Decision Trees	0.6568(0.6520–0.6613) •	0.9568(0.9537–0.9598)	0.9565(0.9532–0.9596)
Boosted DT	0.6817(0.6773–0.6858) •	0.9789(0.9770–0.9806) •	0.9792(0.9774–0.9808) •
SMO	0.6021(0.5986–0.6058) •	0.8301(0.8246–0.8350) •	0.8232(0.8183–0.8278) •

- comparison dataset
∘ statistically significant improvement
• statistically significant degradation

The distributions of the benign samples with positive features for the `AndroidManifest.xml` are not as skewed as the histograms for the `resources.arsc` or `classes.dex` as seen in Figure 4.5. The malicious histogram shows a spike of 145 samples that have 375 of the positive features. All 145 of the samples are in the `Anserverbot` family of malware.

Unlike the other feature sources, the high number of features associated with a specific malware family does not influence the true positive results. The true positive rates for the `AndroidManifest.xml` are much higher as seen in Table 4.9. The feature source has a true positive rate of 91.69% with the boosted decision trees and a 91.97% for the

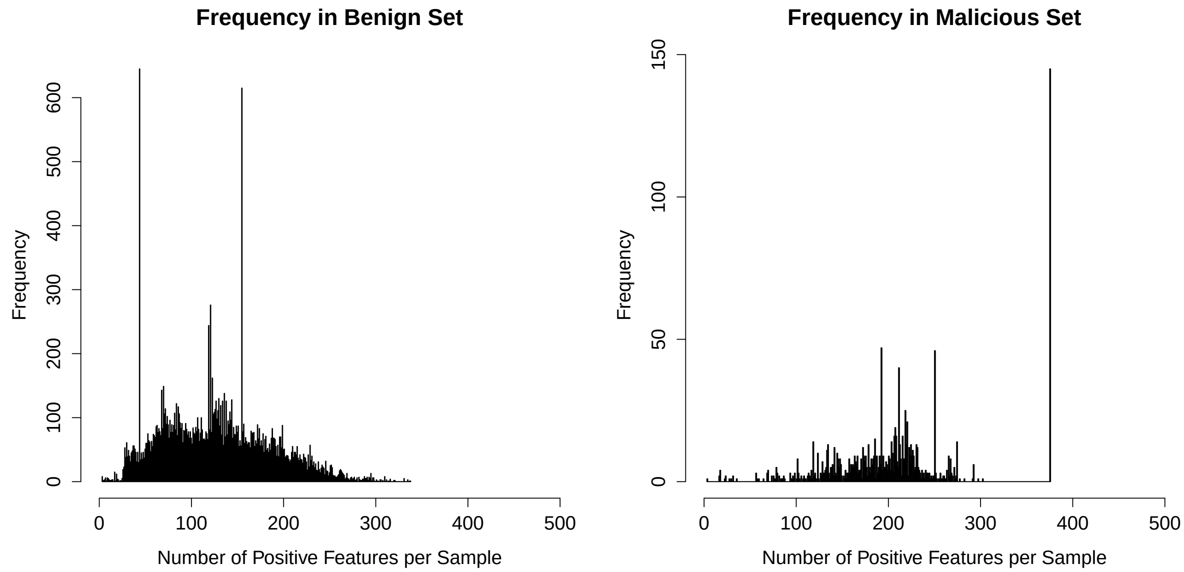

Figure 4.5: Frequency of positive features of datasets for `AndroidManifest.xml`

instance based learner. The high true positive rates and the low false positives show that the `AndroidManifest.xml` is the best feature source in the experiment.

Table 4.9: Confusion matrix rates for AndroidManifest.xml

	Confusion Matrix Rates with 95% CI			
	TPR	FPR	TNR	FNR
Naïve Bayes	0.8153(0.8077–0.8225)	0.07836(0.07742–0.07935)	0.9216(0.9206–0.9225)	0.1846(0.1775–0.1918)
Boosted decision trees	0.9169(0.9119–0.9224)	0.001918(0.001689–0.002126)	0.9980(0.9978–0.9983)	0.08301(0.07752–0.08878)
Boosted Naïve Bayes	0.8726(0.8665–0.8788)	0.02978(0.02737–0.03222)	0.9702(0.9678–0.9724)	0.1273(0.1210–0.1334)
SMO	0.8847(0.8792–0.8903)	0.001055(0.0008890–0.001228)	0.9989(0.9987–0.9991)	0.1152(0.1098–0.1210)
Instance based	0.9197(0.9148–0.9243)	0.007293(0.006890–0.007687)	0.9927(0.9923–0.9930)	0.08023(0.07557–0.08479)
Decision trees	0.9052(0.8995–0.9108)	0.008638(0.008150–0.009135)	0.9913(0.9908–0.9918)	0.09476(0.08861–0.1003)

4.4.4 Instance Based Classifier Results.

The instance based classifier saves each of the samples from the training set and compares each of the test samples to the training samples to find the closest match. Table 4.10 shows that the `resources.arsc` has a low TP rate. The `AndroidManifest.xml` has the highest true positive rate with the instance based classifier. Table 4.11 shows the instance based classifier only classifies on average 345.9 of the 1,260 malicious samples correctly with the `resources.arsc`. The `resources.arsc` has an average accuracy of 0.945 with the 95% confidence interval of 0.9453 to 0.9465. Both of the permission types have the same rates of classification except that the full permissions misclassifies on average 5.5 of the benign samples as malicious.

Table 4.10: Confusion matrix rates for instance based

	Confusion Matrix Rates with 95% CI			
	TPR	FPR	TNR	FNR
Combined	0.8808(0.8758–0.8858)	0.01605(0.01541–0.01666)	0.9839(0.9833–0.9845)	0.1191(0.1136–0.1242)
`classes.dex`	0.7754(0.7684–0.7825)	0.004578(0.004250–0.004914)	0.9954(0.9951–0.9957)	0.2245(0.2173–0.2317)
`AndroidManifest.xml`	0.9197(0.9149–0.9241)	0.007293(0.006887–0.007667)	0.9927(0.9923–0.9930)	0.08023(0.07517–0.08497)
Permissions full	0.8845(0.8791–0.8904)	0.008765(0.008356–0.009158)	0.9912(0.9907–0.9916)	0.1154(0.1101–0.1206)
Permissions tail	0.8844(0.8790–0.8899)	0.008433(0.008023–0.008837)	0.9915(0.9911–0.9919)	0.1155(0.1101–0.1209)
`resources.arsc`	0.2745(0.2669–0.2829)	0.003112(0.002852–0.003371)	0.9968(0.9966–0.9971)	0.7254(0.7168–0.7339)

Table 4.11: Confusion matrix data for instance based with 95% CI

	True Positives	False Positives	True Negatives	False Negatives
Combined	1,109.9(1,109.0977–1,110.9)	266.1(262.5045–269.72)	16,310.9(16,307.2818–16,314.5)	150.1(149.1208–150.83)
`classes.dex`	977.1(974.862–979.81)	75.9(73.0554–78.63)	16,501.1(16,498.2633–16,504.27)	282.9(280.3577–285.37)
`AndroidManifest.xml`	1,158.9(1,156.9857–1,160.66)	120.9(117.9327–123.45)	16,456.1(16,453.7898–16,458.87)	101.1(99.2847–102.98)
Permissions full	1,114.5(1,111.6749–1,117.05)	145.3(143.361–147.6)	16,431.7(16,429.3411–16,433.81)	145.5(143.0471–148.45)
Permissions tail	1,114.4(1,111.4635–1,116.67)	139.8(138.1525–141.78)	16,437.2(16,435.4611–16,438.88)	145.6(143.2455–148.53)
`resources.arsc`	345.9(345.9–346.06)	51.6(50.3145–52.67)	16,525.4(16,524.3199–16,526.56)	914.1(913.9382–914.1)

The Table 4.12 compares the instance based learner performance for each feature source to the performances in the other classifiers. Weka calculates the statistical comparison using a corrected paired t-test with 0.05 significance. The table identifies the comparison class with a symbol next to the confidence interval. The presence of the bullet indicate that naïve Bayes, boosted naïve Bayes, decision trees, and SMO all perform worse than the instance based learner for all feature sources. The only classifier in this test that significantly performs better is the boosted decision trees for `AndroidManifest.xml` and `resources.arsc`.

Table 4.12: ROC area under the curve comparing instance based learner to other classifiers

Dataset	IB*k*	Naïve Bayes	Boosted NB
Combined	0.9808(0.9791–0.9824) -	0.9114(0.9085–0.9140) •	0.9458(0.9432–0.9481) •
`classes.dex`	0.9608(0.9587–0.9629) -	0.8605(0.8564–0.8642) •	0.8351(0.8311–0.8393) •
`AndroidManifest.xml`	0.9841(0.9825–0.9857) -	0.9306(0.9279–0.9331) •	0.9751(0.9733–0.9766) •
`resources.arsc`	0.6738(0.6692–0.6782) -	0.6140(0.6087–0.6187) •	0.6189(0.6142–0.6238) •
Full Permission	0.9806(0.9790–0.9822) -	0.9222(0.9192–0.9247) •	0.9497(0.9471–0.9523) •
Permission Tail	0.9807(0.9790–0.9822) -	0.9240(0.9213–0.9266) •	0.9483(0.9457–0.9507) •

Dataset	Decision Trees	Boosted DT	SMO
Combined	0.9556(0.9530–0.9580) •	0.9751(0.9731–0.9769) •	0.8159(0.8119–0.8201) •
`classes.dex`	0.9458(0.9430–0.9488) •	0.9550(0.9523–0.9575)	0.8139(0.8100–0.8182) •
`AndroidManifest.xml`	0.9634(0.9605–0.9660) •	0.9890(0.9877–0.9902) ◦	0.9419(0.9390–0.9447) •
`resources.arsc`	0.6568(0.6520–0.6613) •	0.6817(0.6773–0.6858) ◦	0.6021(0.5986–0.6058) •
Full Permission	0.9568(0.9537–0.9598) •	0.9789(0.9770–0.9806)	0.8301(0.8246–0.8350) •
Permission Tail	0.9565(0.9532–0.9596) •	0.9792(0.9774–0.9808)	0.8232(0.8183–0.8278) •

- comparison dataset
◦ statistically significant improvement
• statistically significant degradation

4.4.5 Naïve Bayes Results.

The naïve Bayes classifier does not perform as well as the instance based learner. Observing the information independently from the other classifiers the best feature source is still the `AndroidManifest.xml` with the highest true positive rate as seen in Table 4.13. Naïve Bayes has the lowest false positive rate of all of the classifiers. Both the `classes.dex` and the combined file only had seven false positives.

Table 4.13: Confusion matrix rates for naïve Bayes

	Confusion Matrix Rates with 95% CI			
	TPR	FPR	TNR	FNR
Combined	0.3738(0.3656–0.3823)	0.0002412(0.0001684–0.0003240)	0.9997(0.9996–0.9998)	0.6261(0.6183–0.6341)
`classes.dex`	0.3734(0.3657–0.3818)	0.0004222(0.0003316–0.0005160)	0.9995(0.9994–0.9996)	0.6265(0.6183–0.6350)
`AndroidManifest.xml`	0.8153(0.8082–0.8229)	0.07836(0.07740–0.07932)	0.9216(0.9205–0.9226)	0.1846(0.1774–0.1919)
Permissions full	0.6678(0.6583–0.6772)	0.04485(0.04377–0.04596)	0.9551(0.9541–0.9561)	0.3321(0.3229–0.3413)
Permissions tail	0.6980(0.6892–0.7060)	0.04718(0.04608–0.04823)	0.9528(0.9517–0.9539)	0.3019(0.2937–0.3108)
`resources.arsc`	0.2071(0.2006–0.2140)	0.005477(0.005132–0.005867)	0.9945(0.9941–0.9948)	0.7928(0.7855–0.7996)

Even though the `AndroidManifest.xml` has the highest true positive rate of the feature sources, the `classes.dex` and combined feature sources have the highest accuracy as seen in Table 4.14. The high accuracy is due to the classifier's ability to classify the benign samples. The `classes.dex` and the combined files classify the benign samples positively at a rate of 0.9995 and 0.9997. The high true negative rate is from the distribution of features as discussed in 4.4.2. In the Kolter and Maloof experiments the naïve Bayes classifier is the worst performer [29]. Comparing the area under the curve performance naïve Bayes only performs better than the SMO classifier with the exception of the `AndroidManifest.xml`.

Table 4.14: Accuracy for naïve Bayes

	Mean	(95% CI)
Combined	0.9555	(0.955-0.9561)
`classes.dex`	0.9553	(0.9548-0.9559)
`AndroidManifest.xml`	0.9141	(0.9131-0.9152)
Permissions full	0.9348	(0.9335-0.936)
Permissions tail	0.9348	(0.9337-0.936)
`resources.arsc`	0.9389	(0.9383-0.9395)

4.4.6 Boosted Naïve Bayes Results.

The goal of boosting a machine learning algorithm is to minimize the error to obtain better results. The first boosted algorithm in the experiment is boosting naïve Bayes. The accuracies of the boosted naïve Bayes with the `resources.arsc` and the `classes.dex` performs the similar to these features in the unboosted naïve Bayes. The results are the same as seen in Table 4.15.

Table 4.15: Accuracy comparison between naïve Bayes and boosted naïve Bayes

	naïve Bayes	Boosted naïve Bayes
Combined	0.9555(0.955-0.9561)	0.958 (0.9569-0.9592)
`classes.dex`	0.9553(0.9548-0.9559)	0.9553 (0.9548-0.9559)
`AndroidManifest.xml`	0.9141 (0.9131-0.9152)	0.9633 (0.9613-0.9653)
Permissions full	0.9348 (0.9335-0.936)	0.9602 (0.9592-0.9611)
Permissions tail	0.9348 (0.9337-0.936)	0.9603 (0.9595-0.9611)
`resources.arsc`	0.9389 (0.9383-0.9395)	0.9389 (0.9383-0.9395)

Since boosting reduces the error on classifiers, the boosted classifier should perform better. The accuracy increases, but the ROC AUC decreases for the `classes.dex` feature source. The ROC AUC for boosted naïve Bayes is .8351 and .8605 for naïve Bayes. This comparison shows a degradation in the classification. The ROC in Figure 4.6 illustrates the area under the curve for boosted naïve Bayes is lower. The boosted naïve Bayes ROC has a higher false positive rate when covering more true positive samples leads to the difference in ROC values.

Boosting naïve Bayes reduces the number of false positives for the `AndroidManifest.xml` by almost one-third and increases the true positive rate. Considering that, the combined file consists of 95% of the classes.dex source, the true positive results increase from 471 to 714 samples. The number of true positives decreases in the permission sources, but their false positive rate reduces more than one-third from the naïve Bayes.

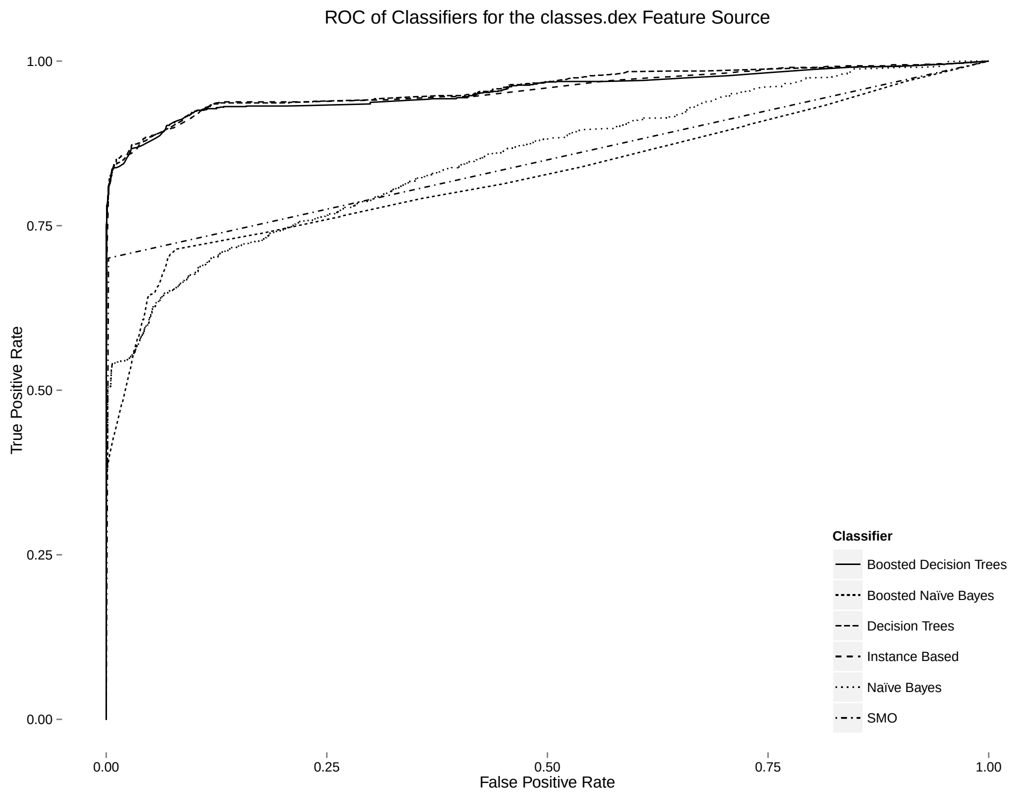

Figure 4.6: ROC of `classes.dex` comparing classifiers

The changes in the true positive and false positive rates change the ROC AUC. The overall best source for boosted naïve Bayes is still the `AndroidManifest.xml` with an AUC of 0.9751. The worst performer is still the `resources.arsc` with only an increase from 0.6140 with naïve Bayes to 0.6189 with boosted naïve Bayes as seen in Table 4.16.

Table 4.16: Comparison of mean ROC for boosted naïve Bayes with 95% CI

	boosted naïve Bayes	naïve Bayes
Combined	0.9458 (0.9432–0.9481)	0.9114 (0.9085–0.9140)
`classes.dex`	0.8351 (0.8311–0.8393)	0.8605 (0.8564–0.8642)
`AndroidManifest.xml`	0.9751 (0.9733–0.9766)	0.9306 (0.9279–0.9331)
`resources.arsc`	0.6189 (0.6142–0.6238)	0.6140 (0.6087–0.6187)
Full Permission	0.9497 (0.9471–0.9523)	0.9222 (0.9192–0.9247)
Permission Tail	0.9483 (0.9457–0.9507)	0.9240 (0.9213–0.9266)

4.4.7 *Decision Trees Results.*

In this experiment, decision trees have a higher ROC AUC than the SMO and naïve Bayes as seen in Table 4.17. The `AndroidManifest.xml` is the best feature source for decision trees, but has a better AUC with the boosted naïve Bayes. The `AndroidManifest.xml` feature source has an AUC of 0.9634 and detects 90.52% of the malicious applications. In previous classifiers, the `AndroidManifest.xml` has the best performance of the feature sources. The permission based feature sources and the `AndroidManifest.xml` have similar accuracies with the means falling within the other's confidence intervals as seen in Table 4.18.

Table 4.17: ROC AUC comparison decision trees, naïve Bayes, and SMO with 95% CI

Dataset	Decision Trees	Naïve Bayes	SMO
Combined	0.9556 (0.9530–0.9580)	0.9114	(0.9085–0.9140) 0.8159 (0.8119–0.8201)
`classes.dex`	0.9458 (0.9430–0.9488)	0.8605 (0.8564–0.8642)	0.8139 (0.8100–0.8182)
`AndroidManifest.xml`	0.9634 (0.9605–0.9660)	0.9306 (0.9279–0.9331)	0.9419 (0.9390–0.9447)
`resources.arsc`	0.6568 (0.6520–0.6613)	0.6140 (0.6087–0.6187)	0.6021 (0.5986–0.6058)
Full Permission	0.9568 (0.9537–0.9598)	0.9222 (0.9192–0.9247)	0.8301 (0.8246–0.8350)
Permission Tail	0.9565 (0.9532–0.9596)	0.9240 (0.9213–0.9266)	0.8232 (0.8183–0.8278)

Table 4.18: Accuracy for decision trees

	Mean	(95% CI)
Combined	0.9834	(0.9829-0.9839)
classes.dex	0.9795	(0.9789-0.9802)
AndroidManifest.xml	0.9853	(0.9847-0.9859)
Permissions full	0.9854	(0.9848-0.9859)
Permissions tail	0.9853	(0.9847-0.9858)
resources.arsc	0.9454	(0.9448-0.946)

4.4.8 *Boosted Decision Trees Results.*

The boosted decision tree is the best performing classifier compared to the other classifiers using the feature sources in this experiment with the exception of `classes.dex` and the combined feature source. In this case the mean AUC for the combined feature source for the instance based classifier is 0.9808 and is only 0.9751 for the boosted decision tree as seen in Table 4.19. As for the `classes.dex` the mean AUC for the instance based classifier is 0.9608 and 0.9550 for boosted trees. The best combination of feature source and classifier in this overall experiment is the `AndroidManifest.xml`with the boosted decision tree.

Table 4.19: ROC AUC comparison of decision trees, boosted DT, and IB*k* with 95% CI

Dataset	Decision Trees	Boosted Decision Trees	IB*k*
Combined	0.9556 (0.9530–0.9580)	0.9751 (0.9731–0.9769)	0.9808 (0.9791–0.9824)
`classes.dex`	0.9458 (0.9430–0.9488)	0.9550 (0.9523–0.9575)	0.9608 (0.9587–0.9629)
`AndroidManifest.xml`	0.9634 (0.9605–0.9660)	0.9890 (0.9877–0.9902)	0.9841 (0.9825–0.9857)
`resources.arsc`	0.6568 (0.6520–0.6613)	0.6817 (0.6773–0.6858)	0.6738 (0.6692–0.6782)
Full Permission	0.9568 (0.9537–0.9598)	0.9789 (0.9770–0.9806)	0.9806 (0.9790–0.9822)
Permission Tail	0.9565 (0.9532–0.9596)	0.9792 (0.9774–0.9808)	0.9807 (0.9790–0.9822)

The `AndroidManifest.xml` has a true positive rate of 0.9169 with the boosted decision tree, which is not as successful as the true positive rate from the instance based learner with 0.9197. Even though the instance based learner has a higher true positive rate, the overall accuracy of the boosted decision tree with `AndroidManifest.xml` is 0.9924 which is higher than the accuracy of the instance based classifier with 0.9876 as seen in Table 4.20. Unlike the comparison between naïve Bayes and boosted naïve Bayes, the boosting of the decision trees improves the classifier.

Table 4.20: Mean accuracies of decision trees, boosted DT, and IBk with 95% CI

	Decision Trees	Boosted Decision Trees	IBk
Combined	0.9834 (0.9829-0.9839)	0.9881(0.9876-0.9885)	0.9767 (0.976-0.9773)
classes.dex	0.9795 (0.9789-0.9802)	0.9831 (0.9827-0.9836)	0.9799 (0.9794-0.9805)
AndroidManifest.xml	0.9853 (0.9847-0.9859)	0.9924 (0.9919-0.9928)	0.9876 (0.987-0.988)
Permissions full	0.9854 (0.9848-0.9859)	0.9887 (0.9883-0.9892)	0.9837 (0.9831-0.9842)
Permissions tail	0.9853(0.9847-0.9858)	0.9888 (0.9883-0.9893)	0.984 (0.9835-0.9845)
resources.arsc	0.9454 (0.9448-0.946)	0.9454 (0.9448-0.946)	0.9459 (0.9452-0.9464)

4.4.9 SMO Results.

The last classifier in this experiment is the SMO, which is an optimization of the SVM algorithm.SMO did not perform as well as the other classifiers in this experiment. Only the `AndroidManifest.xml` feature source has a true positive rate above 0.80 with a true positive rate of 0.8874 as seen in Table 4.21. The low true positive rate for the `resources.arsc` is due to the features in the `resources.arsc` favoring a specific family of malware as explained in 4.4.1.

Table 4.21: Confusion matrix rates for SMO

Confusion Matrix Rates with 95% CI

	TPR	FPR	TNR	FNR
Combined	0.6319(0.6245–0.6394)	0.0001206(7.506e-05–0.0001742)	0.9998(0.9998–0.9999)	0.3680(0.3597–0.3761)
classes.dex	0.6279(0.6204–0.6354)	0.0001206(7.509e-05–0.0001667)	0.9998(0.9998–0.9999)	0.3720(0.3640–0.3799)
AndroidManifest.xml	0.8847(0.8792–0.8903)	0.001055(0.0008890–0.001228)	0.9989(0.9987–0.9991)	0.1152(0.1098–0.1210)
Permissions full	0.6665(0.6548–0.6766)	0.006352(0.005976–0.006713)	0.9936(0.9932–0.9940)	0.3334(0.3233–0.3440)
Permissions tail	0.6525(0.6421–0.6620)	0.006044(0.005660–0.006400)	0.9939(0.9935–0.9943)	0.3474(0.3373–0.3573)
resources.arsc	0.2042(0.1975–0.2110)	0.0(0.0–0.0)	1.000(0.0–0.0)	0.7957(0.7892–0.8028)

The highest parsing method is the `AndroidManifest.xml` with an AUC of 0.935. The lowest performing parsing method is the `resources.arsc` with an AUC of 0.616 as seen in Table 4.22. In the Kolter and Maloof experiments, SMO is the second best classifier with large data sets [29]. In this experiment, SMO is the worst performing classifer.

Table 4.22: ROC AUC for SMO classifier

Dataset	Mean	(95% CI)
Combined	0.8159	(0.8119–0.8201)
`classes.dex`	0.8139	(0.8100–0.8182)
`AndroidManifest.xml`	0.9419	(0.9390–0.9447)
`resources.arsc`	0.6021	(0.5986–0.6058)
Full Permission	0.8301	(0.8246–0.8350)
Permission Tail	0.8232	(0.8183–0.8278)]

4.5 Summary

The best feature source/classifier pair in this experiment is the `AndroidManifest.xml` with boosted decision trees. This classification pair classifies with high accuracy rates and ROC AUC. The mean of the ROC AUC is 0.9890 with a 95% confidence interval between 0.9877 and 0.9902 and the accuracy is 99.24% with a 95% confidence interval of 99.19% to 99.28%.

The worst feature source in this set of experiments is the `resources.arsc`. Looking at the information gain levels from Section 4.3, the top *n*-grams have low information gain and almost constant information gain for all 500 selected *n*-grams. The `resources.arsc` performs poorly compared to the other feature sources when applied to the classifiers. The feature source's highest true positive rate is never higher than 0.2753 with the decision trees. The highest mean AUC for the `resources.arsc` is never higher than 0.6817. These poor results may be due to the `resources.arsc` features belonging to a specific malware family.

The hypothesis is that the `classes.dex` and the combined feature source are the highest performers. Since the `classes.dex` contains the byte code for the application to operate, the hypothesis includes that this method would be a good feature source for classification decisions. The top n-grams from each source are in the combined file, but this was 95.8% of the *n*-grams in the top 500 *n*-grams of the `classes.dex` files. From this, the hypothesis is that the combined file performs similar to the classes.dex. In all classifiers, the combined feature source performs better than the `classes.dex`. The 19 *n*-grams from the `AndroidManifest.xml` increase the performance of the feature source.

The best performing feature sources in this experiment are the `AndroidManifest.xml` and the permission-based methods. Many of the services that the Android malware exploits attempt to access services that require permissions. The permissions with the highest information gain values associate with SMS and `RECEIVE_BOOT_COMPLETED`. Through chaining the *n*-grams in the `AndroidManifest.xml` these same permissions have high information gain values. These three feature sources performed the best during classification. They performed well with the instance based learner and with boosted decision trees. According to Zhou, malware has a high tendency to use these permissions more than benign applications [50].

V. Conclusion

The primary goal of this research is to classify Android applications as either malicious or benign. Achieving this goal requires identification of a salient feature source to perform the classification. This research examines the effectiveness of *n*-grams on a variety of Android application feature sources. This experiment uses the *n*-grams features with the highest information gain values as the selected feature set. In addition to these sources, the permissions of the applications are also in the experiments.

This experiment identifies the `AndroidManifest.xml` as the best feature source compared to the other selected feature sources. The secondary goal of this research is to identify a classifier that produces the highest accuracy and ROC AUC performance metrics. Through testing six classifiers with the six feature sources produces 36 different experiments to compare. In the 36 experiments, the Android manifest file with the boosted decision trees returns the highest ROC AUC of 0.9890 and a mean accuracy of 99.24%.

The `AndroidManifest.xml` file is smaller than other files in the application such as the `classes.dex`. This smaller size allows the feature extractor to parse through the file for textitn-grams quickly. The `AndroidManifest.xml` performs better than the hypothesis of the combined feature source.

The last goal of the research is to identify if sequences of *n*-grams are associated more with malware than other *n*-grams. Analysis of the `AndroidManifest.xml` *n*-grams show that the sequences of *n*-grams are associated with the permissions and actions of malware samples. Analysis shows that the `BOOT_COMPLETED` was found in multiple malware samples.

5.1 Contributions

This research contributes by identifying a feature source within the Android application to classify samples with high rates. The use of *n*-grams to create features from the `AndroidManifest.xml` file performs better than permission based feature selection. The feature source did not just work for one classifier, but provides high classification rates with both boosted decision trees and the instance based classifier.

5.2 Future Work

A limitation on this research is the low number of malicious samples. At the beginning of this research, large data sets of Android malware samples did not exist. The Malware Genome project released a large malicious sample set that allowed for this research to occur [50]. Until this point, researchers had to create their own samples or test with the small sets available [45, 49]. A recommendation for future work includes continuing the research as the number of known malware sample increases.

Possible future work includes implementing this research during application distribution to prevent users from downloading malicious applications to their devices. In addition, other implementations of this research may detect malicious applications directly on the device. Training requires a large amount of memory that is not available to Android devices. The retraining of new samples may occur off the device while detection occurs on the device.

One of the feature sources in this experiment is a combined file with the *n*-grams with the highest information gain values from the three file based sources. A majority of the *n*-grams in this feature source are from the `classes.dex` and no features from the `resources.arsc`. Other combinations of feature sources may provide a better classification rate.

Appendix A: Classifier and Feature Source ROC Graphs

Figures A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.7, A.8, A.9, A.10, A.11, and A.12 are the ROC curve graphs for the classifiers and feature sources in Section 4.4.

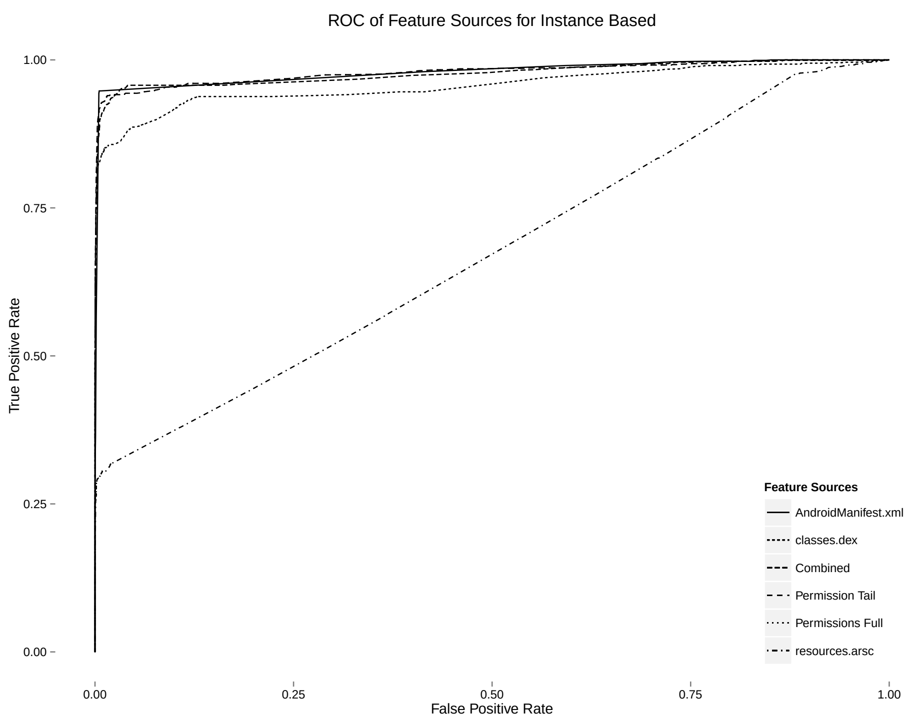

Figure A.1: ROC curve for instance based learner

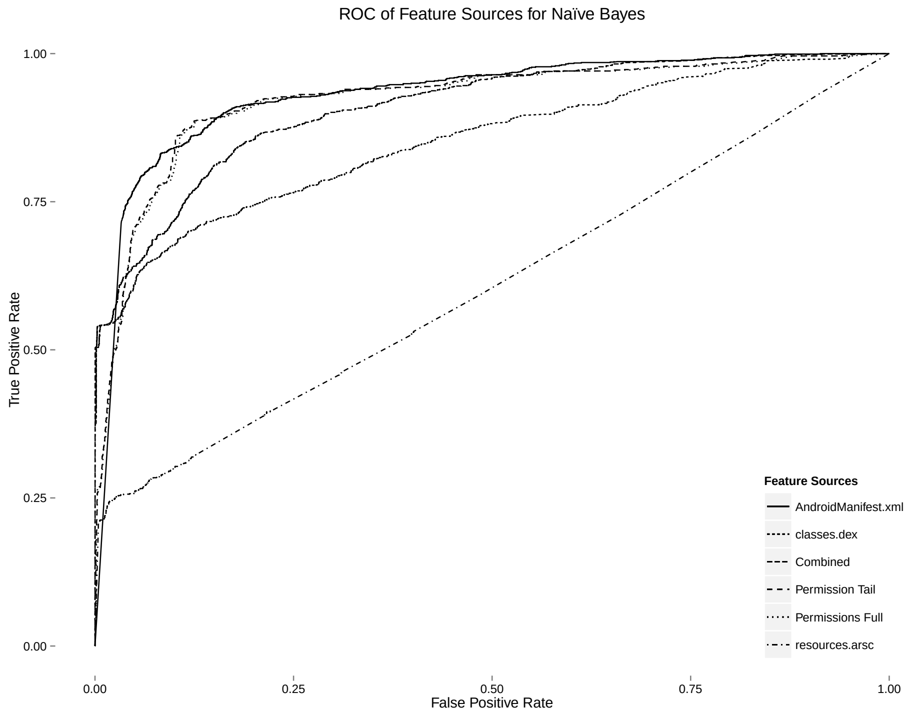

Figure A.2: ROC curve for naïve Bayes

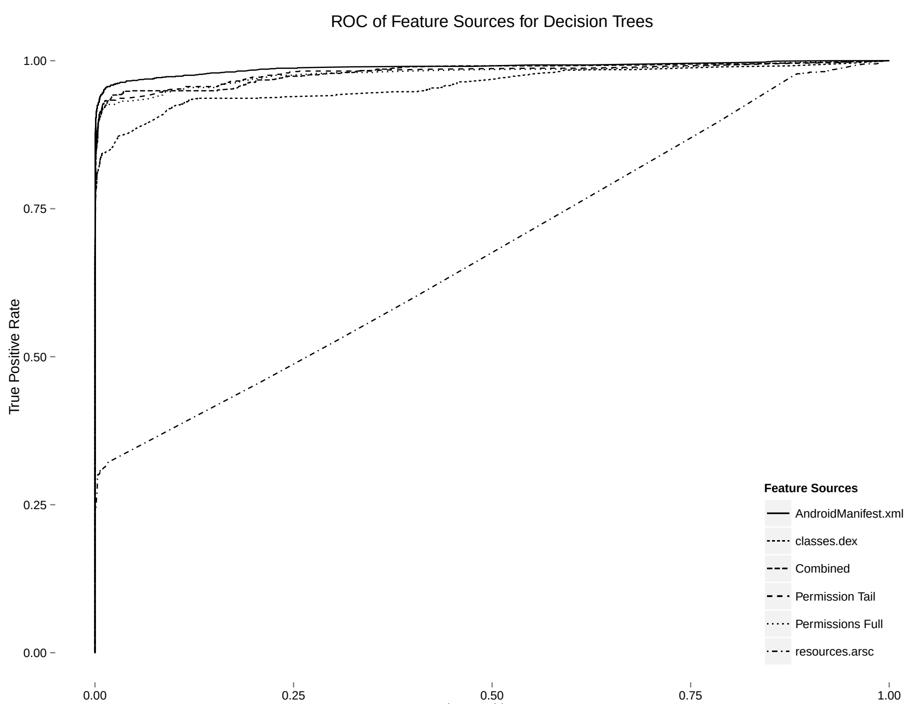

Figure A.3: ROC curve for decision trees

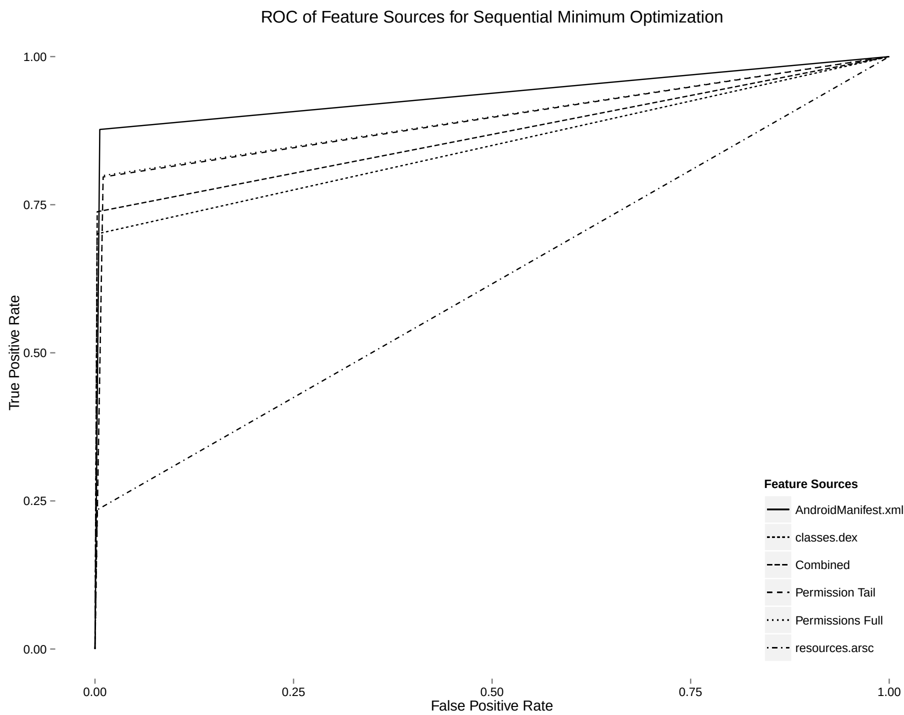

Figure A.4: ROC curve for SMO

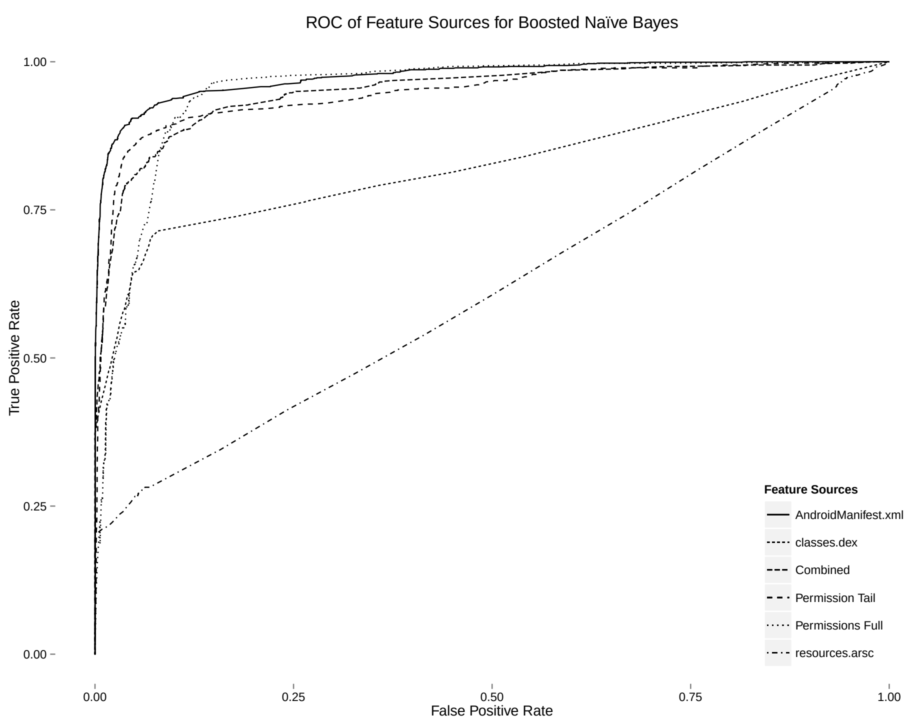

Figure A.5: ROC curve for boosted naïve Bayes

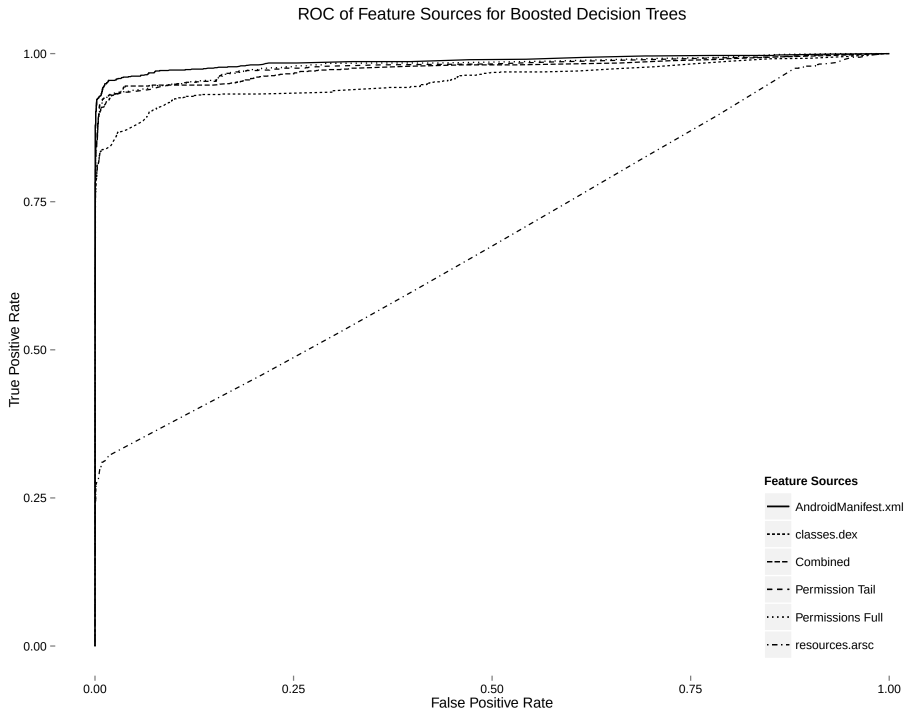

Figure A.6: ROC curve for boosted decision trees

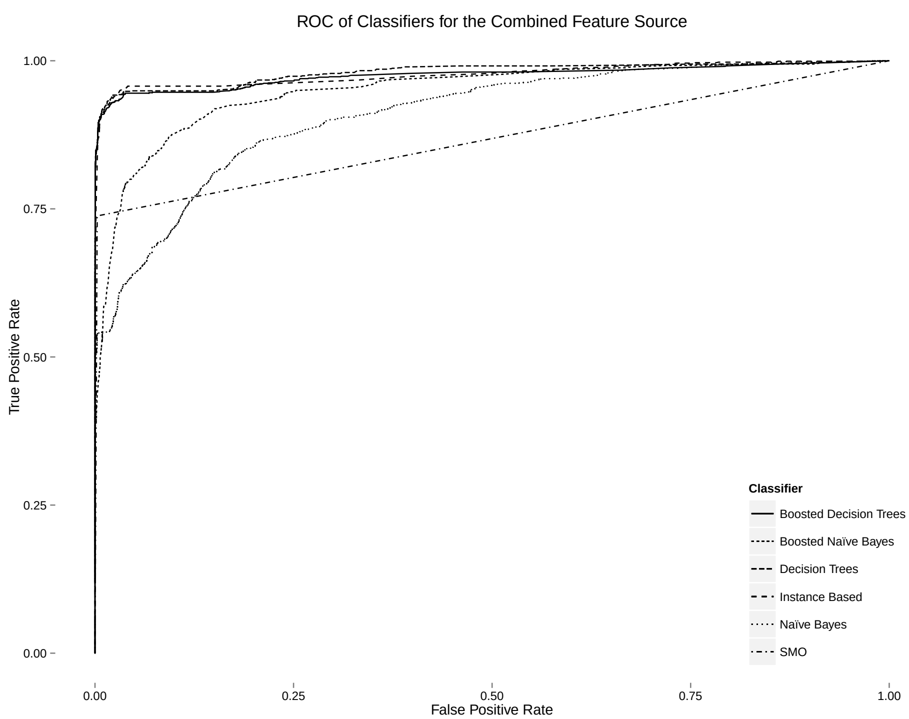

Figure A.7: ROC curve for the combined feature source

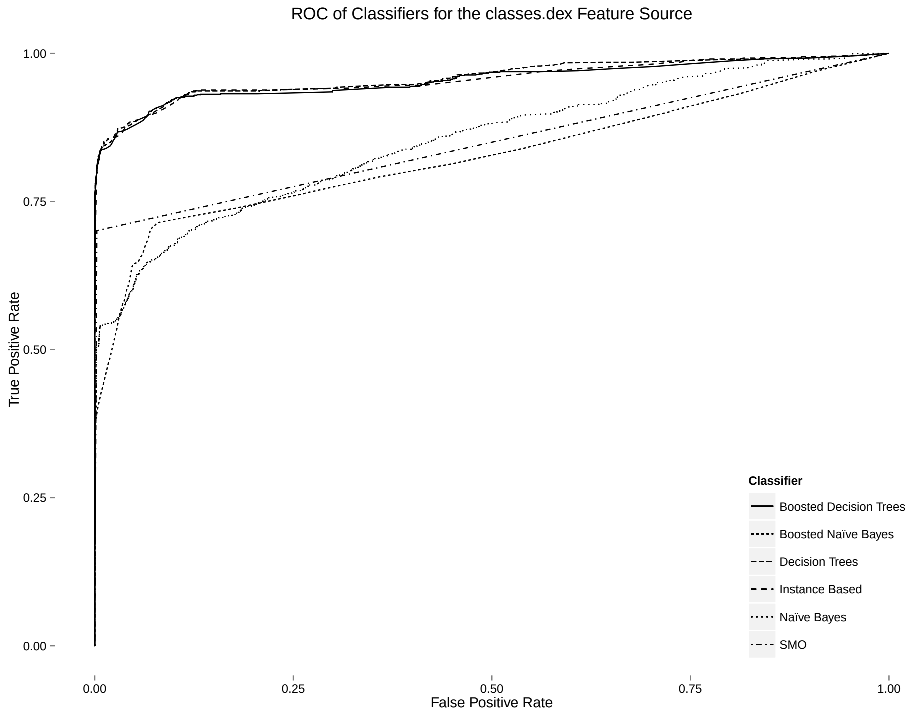

Figure A.8: ROC curve for the `classes.dex` feature source

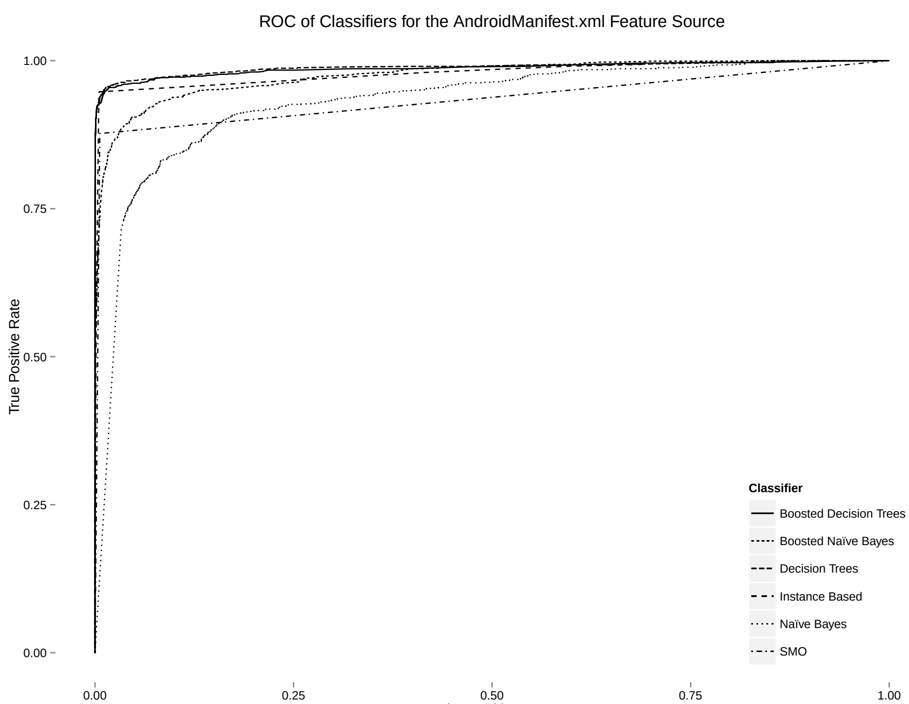

Figure A.9: ROC curve for the `AndroidManifest.xml` feature source

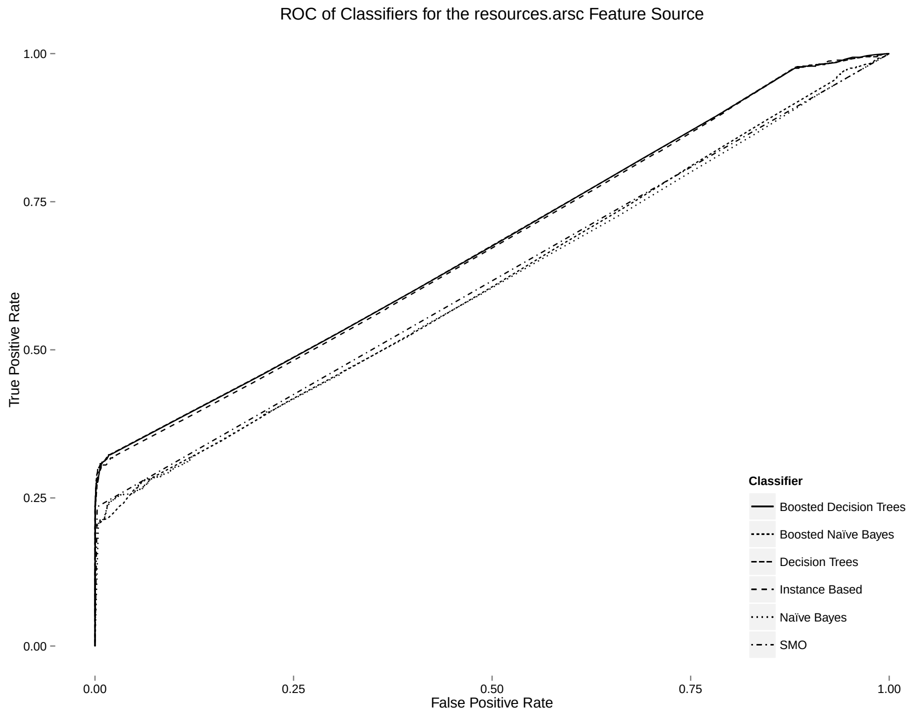

Figure A.10: ROC curve for the `resources.arsc` feature source

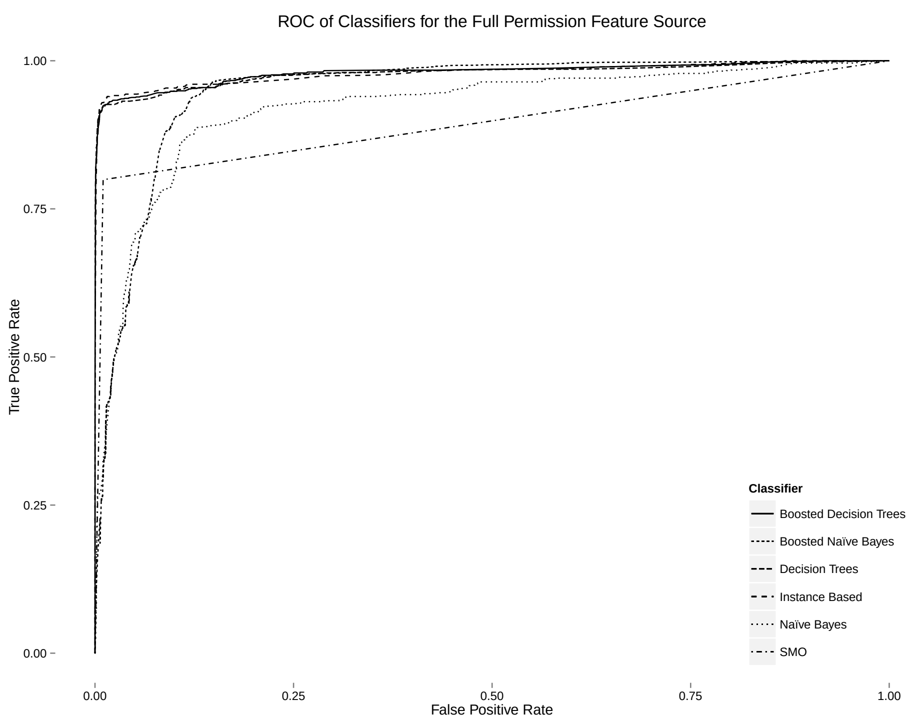

Figure A.11: ROC curve for the full permissions feature source

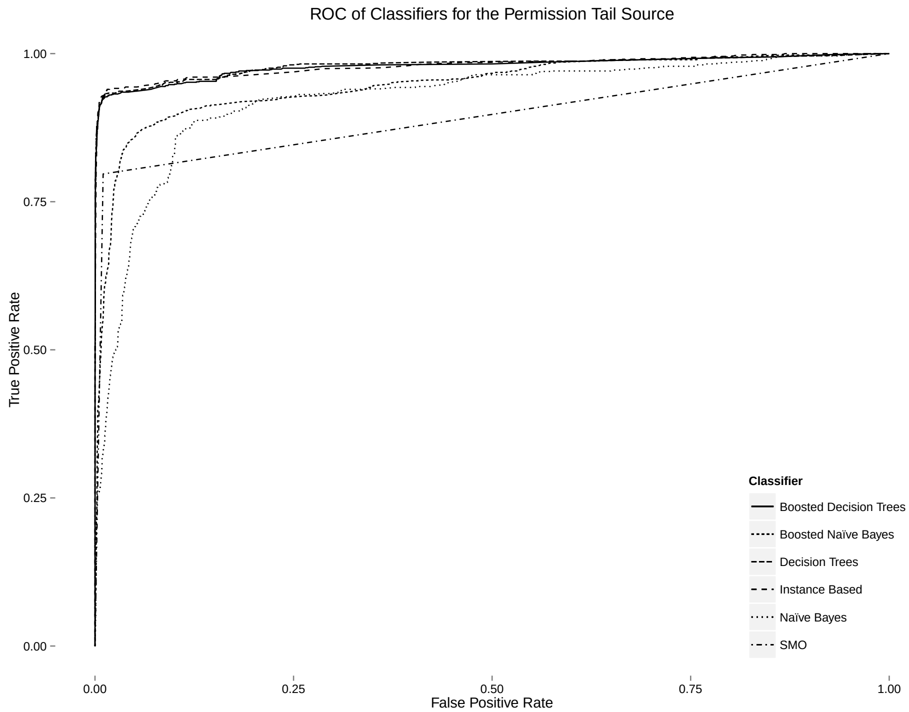

Figure A.12: ROC curve for the permissions tail feature source

Appendix B: Confusion Matrix Data for Classifiers and Feature Sources

Tables B.1, B.2, B.3, B.4, and B.5 are the confusion matrix data results for each of the classifiers in Section 4.4.

Table B.1: Confusion matrix data for naïve Bayes with 95% confidence intervals

	True Positives	False Positives	True Negatives	False Negatives
Combined	471.0(0.0–0.0)	4.0(0.0–0.0)	16,573.0(0.0–0.0)	789.0(0.0–0.0)
`classes.dex`	470.5(470.1646–470.84)	7.0(6.2633–7.67)	16,570.0(16,569.3326–16,570.74)	789.5(789.1646–789.84)
`AndroidManifest.xml`	1,027.3(1,026.2327–1,028.43)	1,299.0(1,297.4187–1,300.45)	15,278.0(15,276.7415–15,279.61)	232.7(231.5673–233.89)
Permissions full	841.5(839.8904–842.97)	743.6(740.5617–745.91)	15,833.4(15,831.2426–15,835.99)	418.5(417.0149–420.14)
Permissions tail	879.5(877.8301–881.11)	782.1(780.0329–784.0)	15,794.9(15,792.957–15,797.04)	380.5(378.8833–382.15)
`resources.arsc`	261.0(0.0–0.0)	90.8(90.6845–91.07)	16,486.2(16,485.9317–16,486.32)	999.0(0.0–0.0)

Table B.2: Confusion matrix data for boosted naïve Bayes with 95% confidence intervals

	True Positives	False Positives	True Negatives	False Negatives
Combined	703.9(679.7699–727.44)	192.5(178.9837–207.66)	16,384.5(16,370.3888–16,398.88)	556.1(531.2016–580.41)
`classes.dex`	470.5(470.1646–470.84)	7.0(6.2633–7.61)	16,570.0(16,569.3292–16,570.74)	789.5(789.1646–789.84)
`AndroidManifest.xml`	1,099.6(1,094.4357–1,104.8)	493.8(467.9354–526.69)	16,083.2(16,049.9685–16,108.05)	160.4(154.8581–165.35)
Permissions full	817.9(806.9329–827.86)	268.1(263.7766–272.26)	16,30,8.9(16,304.9997–16,313.27)	442.1(431.8123–451.4)
Permissions tail	825.4(820.1378–831.11)	274.3(267.6991–280.73)	16,302.7(16,296.1963–16,309.39)	434.6(427.5188–440.5)
`resources.arsc`	261.0(0.0–0.0)	90.8(90.6845–91.07)	16,486.2(16,485.9317–16,486.32)	999.0(0.0–0.0)

Table B.3: Confusion matrix data for decision trees with 95% confidence intervals

	True Positives	False Positives	True Negatives	False Negatives
Combined	1,128.0(1,124.71–1,131.38)	163.5(159.5745–168.2)	16,413.5(16,408.5037–16,418.12)	132.0(128.3806–135.28)
`classes.dex`	1,002.0(998.8636–1,005.21)	107.6(103.6203–111.79)	16,469.4(16,464.841–16473.44)	258.0(255.0396–261.35)
`AndroidManifest.xml`	1,140.6(1,137.1514–1,145.2)	143.2(136.7714–149.88)	16,433.8(16,426.6187–16,439.23)	119.4(114.6722–122.99)
Permissions full	1,123.0(1,121.1932–1,124.62)	124.3(119.4518–129.9)	16,452.7(16,447.2976–16,457.26)	137.0(135.2787–138.92)
Permissions tail	1,124.2(1,121.2991–1,125.86)	127.0(123.4571–132.53)	16,450.0(16,444.8617–16,453.6)	135.8(134.1627–138.64)
`resources.arsc`	347.0(344.3466–348.98)	60.1(56.9252–62.98)	16,516.9(16,514.263–16,519.9)	913.0(910.8268–915.5)

Table B.4: Confusion matrix data for boosted decision trees with 95% confidence intervals

	True Positives	False Positives	True Negatives	False Negatives
Combined	1,121.2(1,117.4387–1,124.6)	74.2(70.1141–79.02)	16,502.8(16,498.0996–16,507.04)	138.8(135.8226–142.31)
`classes.dex`	1,024.6(1,023.0308–1,025.94)	65.7(60.8121–70.52)	16,511.3(16,506.0899–16,516.73)	235.4(234.1389–236.97)
`AndroidManifest.xml`	1,155.4(1,152.7209–1,158.14)	31.8(29.502–34.01)	16,545.2(16,542.99–16,547.6)	104.6(101.933–107.42)
Permissions full	1,138.3(1,135.3724–1,140.79)	79.2(76.6101–81.43)	16,497.8(16,495.5934–16,500.4)	121.7(119.2144–124.76)
Permissions tail	1,137.7(1,132.451–1,140.85)	77.9(73.1983–82.53)	16,499.1(16,494.6218–16,503.56)	122.3(118.771–126.95)
`resources.arsc`	347.0(344.3466–348.96)	60.1(57.2897–62.69)	16,516.9(16,514.3651–16,519.85)	913.0(910.9073–915.65)

Table B.5: Confusion matrix data for SMO with 95% confidence intervals

	True Positives	False Positives	True Negatives	False Negatives
Combined	796.3(793.5084–798.74)	2.0(0.0–0.0)	16,575.0(0.0–0.0)	463.7(461.3317–466.39)
`classes.dex`	791.2(787.5885–793.58)	2.0(0.0–0.0)	16,575.0(0.0–0.0)	468.8(466.2285–472.55)
`AndroidManifest.xml`	1,114.8(1,113.1085–1,116.54)	17.5(16.1249–18.67)	16,559.5(16,558.2506–16,560.83)	145.2(143.4189–146.88)
Permissions full	839.8(836.7783–844.43)	105.3(102.4681–108.12)	16,471.7(16,469.1739–16,474.4)	420.2(415.9707–423.25)
Permissions tail	822.2(819.0974–825.5)	100.2(98.0826–103.02)	16,476.8(16,474.4088–16,479.16)	437.8(434.1522–441.18)
`resources.arsc`	257.3(257.0098–257.55)	0.0(0.0–0.0)	16,577.0(0.0–0.0)	1,002.7(1,002.4528–1002.99)

Bibliography

[1] Abou-Assaleh, T., N. Cercone, V. Keselj, and R. Sweidan. "N-gram-based detection of new malicious code". *Proceedings of the 28th Annual International Computer Software and Applications Conference*, 41–42. IEEE, 2004.

[2] Aha, D.W., D. Kibler, and M.K. Albert. "Instance-based learning algorithms". *Machine learning*, 6(1):37–66, 1991.

[3] APKTOP. "Free Android Apps, Games Download From Android Market", August 2012. URL http://www.iapktop.com. Accessed Aug. 24, 2012.

[4] Asrar, I. "Android.Counterclank Found in Official Android Market", Jan 2012. URL http://www.symantec.com/connect/fr/blogs/androidcounterclank-found-official-android-market. Accessed Apr. 16, 2012.

[5] Beresford, A.R., A. Rice, N. Skehin, and R. Sohan. "MockDroid: trading privacy for application functionality on smartphones". *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, 49–54. ACM, 2011.

[6] Bläsing, T., L. Batyuk, A.D. Schmidt, S.A. Camtepe, and S. Albayrak. "An Android application sandbox system for suspicious software detection". *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, 55–62. IEEE, 2010.

[7] Bose, A., X. Hu, K.G. Shin, and T. Park. "Behavioral detection of malware on mobile handsets". *Proceedings of the 6th international conference on Mobile systems, applications, and services*, 225–238. ACM, 2008.

[8] Bose, A. and K.G. Shin. "On mobile viruses exploiting messaging and bluetooth services". *Securecomm and Workshops*, 1–10. IEEE, 2006.

[9] Burguera, I., U. Zurutuza, and S. Nadjm-Tehrani. "Crowdroid: behavior-based malware detection system for Android". *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 15–26. ACM, 2011.

[10] Cheng, J., S.H.Y. Wong, H. Yang, and S. Lu. "Smartsiren: virus detection and alert for smartphones". *Proceedings of the 5th international conference on Mobile systems, applications and services*, 258–271. ACM, 2007.

[11] Dagon, D., T. Martin, and T. Starner. "Mobile phones as computing devices: The viruses are coming!" *Pervasive Computing, IEEE*, 3(4):11–15, 2004.

[12] Enck, W., P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. "TaintDroid: an information-flow tracking system for realtime privacy monitoring

on smartphones". *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 1–6. 2010.

[13] Enck, W., D. Octeau, P. McDaniel, and S. Chaudhuri. "A study of android application security". *Proceedings of the 20th USENIX Security Symposium.*

[14] Enck, W., M. Ongtang, and P. McDaniel. "On lightweight mobile phone application certification". *Proceedings of the 16th ACM conference on Computer and communications security*, 235–245. ACM, 2009.

[15] Felt, A.P., E. Chin, S. Hanna, D. Song, and D. Wagner. "Android permissions demystified". *Proceedings of the 18th ACM conference on Computer and communications security*, 627–638. ACM, 2011.

[16] Felt, A.P., M. Finifter, E. Chin, S. Hanna, and D. Wagner. "A survey of mobile malware in the wild". *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 3–14. ACM, 2011.

[17] Fleizach, C., M. Liljenstam, P. Johansson, G.M. Voelker, and A. Mehes. "Can you infect me now?: malware propagation in mobile phone networks". *Proceedings of the 2007 ACM workshop on Recurring malcode*, 61–68. ACM, 2007.

[18] Freund, Y. and R.E. Schapire. "Experiments with a New Boosting Algorithm". *Proceedings of the 11th International Conference on Machine Learning*, 148–156. 1996.

[19] Google. "Android Open Source Project", 2007. URL http://source.android.com/about/index.html. Accessed Dec. 14, 2012.

[20] Google. "Android SDK", June 2012. URL http://developer.android.com/index.html. Accessed Dec. 19, 2012.

[21] Google. "Application Fundamentals - Android Developers", December 2012. URL http://developer.android.com/guide/components/fundamentals.html. Accessed Dec. 19, 2012.

[22] Google. "Google Play", Aug 2012. URL https://play.google.com/store. Accessed Aug. 8, 2012.

[23] Google. "Dashboards - Android Developers", January 2013. URL http://developer.android.com/about/dashboards/index.html. Accessed Jan. 14, 2013.

[24] Guo, C., H.J. Wang, and W. Zhu. "Smart-phone attacks and defenses". *Proceedings of Third ACM Workshop on Hot Topics in Networks*. 2004.

[25] Hall, M., E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. "The WEKA data mining software: an update". *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[26] Hornyack, P., S. Han, J. Jung, S. Schechter, and D. Wetherall. "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications". *Proceedings of the 18th ACM conference on Computer and communications security*, 639–652. ACM, 2011.

[27] Jiang, X. "An Evaluation of the Application Verification Service in Android 4.2", Dec 2012. URL http://www.cs.ncsu.edu/faculty/jiang/appverify. Accessed Dec. 28, 2012.

[28] John, G.H. and P. Langley. "Estimating Continuous Distributions in Bayesian Classifiers". *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, 338–345. Morgan Kaufmann Publishers Inc., 1995.

[29] Kolter, J.Z. and M.A. Maloof. "Learning to detect and classify malicious executables in the wild". *The Journal of Machine Learning Research*, 7:2721–2744, 2006.

[30] Leavitt, N. "Mobile phones: the next frontier for hackers?" *Computer*, 38(4):20–23, 2005.

[31] Liu, L., G. Yan, X. Zhang, and S. Chen. "Virusmeter: Preventing your cellphone from spies". *Recent Advances in Intrusion Detection*, 244–264. Springer, 2009.

[32] Lockheimer, H. "Android and Security", Feb 2012. URL http://googlemobile.blogspot.com/2012/02/android-and-security.html. Accessed Dec. 28, 2012.

[33] Mobile, Contagio. "Mobile Malware Mini Dump", Mar 2012. URL http://contagiominidump.blogspot.com/. Accessed Mar. 29, 2012.

[34] Nauman, M., S. Khan, and X. Zhang. "Apex: extending android permission model and enforcement with user-defined runtime constraints". *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 328–332. ACM, 2010.

[35] NDUOA. "Safest Android Games, Android Software Download Station", July 2012. URL http://nduoa.com. Accessed Jul. 31, 2012.

[36] Ongtang, M., S. McLaughlin, W. Enck, and P. McDaniel. "Semantically rich application-centric security in Android". *Security and Communication Networks*, 5(6):658–673, 2011.

[37] Park, Y., C.H. Lee, C. Lee, J.H. Lim, S. Han, M. Park, and S.J. Cho. "RGBDroid: a novel response-based approach to android privilege escalation attacks". *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats*.

[38] Platt, J. "Fast Training of Support Vector Machines Using Sequential Minimal Optimization. Advances in Kernel MethodsSupport Vector Learning (pp. 185–208)". *AJ, MIT Press, Cambridge, MA*, 1999.

[39] Quinlan, J.R. *C4. 5: Programs for Machine Learning*, volume 1. Morgan kaufmann, 1993.

[40] Racic, R., D. Ma, and H. Chen. "Exploiting MMS vulnerabilities to stealthily exhaust mobile phone's battery". *Securecomm and Workshops, 2006*, 1–10. IEEE, 2006.

[41] Russell, S. and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002. ISBN 0137903952.

[42] Schmidt, A.D., R. Bye, H.G. Schmidt, J. Clausen, O. Kiraz, K.A. Yuksel, S.A. Camtepe, and S. Albayrak. "Static analysis of executables for collaborative malware detection on android". *Communications, 2009. ICC'09. IEEE International Conference on*, 1–5. IEEE, 2009.

[43] Schmidt, A.D., J.H. Clausen, A. Camtepe, and S. Albayrak. "Detecting symbian os malware through static function call analysis". *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, 15–22. IEEE, 2009.

[44] Shabtai, A., Y. Fledel, and Y. Elovici. "Automated Static Code Analysis for Classifying Android Applications Using Machine Learning". *Computational Intelligence and Security (CIS), 2010 International Conference on*, 329–333. IEEE, 2010.

[45] Shabtai, A., U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. "Andromaly: a behavioral malware detection framework for android devices". *Journal of Intelligent Information Systems*, 38:1–30, 2012.

[46] Varghese, V. "Dissecting Andro Malware", Sep 2011. URL http://www.sans.org/reading_room/whitepapers/malicious/dissecting-andro-malware_33754. Accessed June. 7, 2012.

[47] VirusTotal. "VirusTotal - Free Online Virus, Malware and URL Scanner", October 2012. URL https://www.virustotal.com. Accessed Oct. 7, 2012.

[48] Witten, I.H., E. Frank, and M.A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2011.

[49] Wu, Dong-Jie, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. "DroidMat: Android Malware Detection through Manifest and API Calls Tracing". *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, 62–69. Aug 2012.

[50] Zhou, Y. and X. Jiang. "Dissecting android malware: Characterization and evolution". *Security and Privacy (SP), 2012 IEEE Symposium on*, 95–109. IEEE, 2012.

[51] Zhou, Y., Z. Wang, W. Zhou, and X. Jiang. “Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets”. *Proc. of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. 2012.

[52] Zhou, Y., X. Zhang, X. Jiang, and V. Freeh. “Taming information-stealing smartphone applications (on Android)”. *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, 93–107, 2011.

[53] Zhu, J., Z. Guan, Y. Yang, L. Yu, H. Sun, and Z. Chen. “Permission-Based Abnormal Application Detection for Android”. *Proceedings of the 14th International Conference on Information and Communications Security*, 228–239, 2012.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*
21–03–2013

2. REPORT TYPE
Master's Thesis

3. DATES COVERED *(From — To)*
Oct 2011–Mar 2013

4. TITLE AND SUBTITLE
Examining Application Components
to Reveal Android Malware

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Guptill, John B., First Lieutenant, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT-ENG-13-M-19

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Center for Cyberspace Research
Attn: Dr. Harold Arata
2950 Hobson Way
WPAFB, OH 45433-7765
(937) 255-3636 ext. 7105 harold.arata@afit.edu

10. SPONSOR/MONITOR'S ACRONYM(S)
AFIT/CCR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A.
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES
This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT
Smartphones are becoming ubiquitous in everyday life and malware is exploiting these devices. Therefore, a means to identify the threats of malicious applications is necessary. This paper presents a method to classify and analyze Android malware through application component analysis.
The experiment parses select portions from Android packages to collect features using byte sequences and permissions of the application. Multiple machine learning algorithms classify the samples of malware based on these features. The experiment utilizes instance based learner, naïve Bayes, decision trees, sequential minimal optimization, boosted naïve Bayes, and boosted decision trees to identify the best components that reveal malware characteristics.
The best case classifies malicious applications with an accuracy of 99.24% and an area under curve of 0.9890 utilizing boosted decision trees. This method does not require scanning the entire application and provides high true positive rates. This thesis investigates the components to provide malware classification.

15. SUBJECT TERMS
n-gram, Android, malware detection, machine learning

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
U	UU	U	U	86	Thomas E. Dube, Maj, USAF
					19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4613 thomas.dube@afit.edu